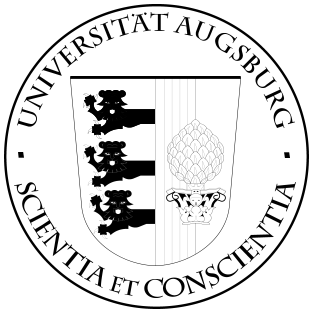
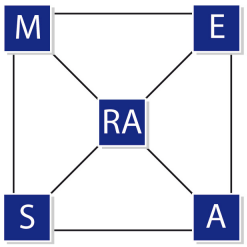
Universität Augsburg



Final System-Level Software for the MERASA Processor

Julian Wolf, Florian Kluge, Irakli Guliashvili

Report 2010-08 Oktober 2010



Institut für Informatik

D-86135 Augsburg

Contents

Abstract

The MERASA[1] project develops hard real-time capable multi-core processor designs with special focus on analysability. The dedicated system software provides basic functionalities of a real-time operating system and enables the execution of application software on the processor.

This report summarises the final MERASA system-level software developed for the MERASA SystemC Simulator and FPGA prototype. We present requirements for a hard real-time capable operating system for embedded multi-cores and the transfer to the implemented MERASA processor prototype showing details on thread management, dynamic memory management and resource management.

1. Introduction

The MERASA project [9] aims to develop multi-core processor designs for hard real-time embedded systems hand in hand with timing analysis techniques to ensure the analysability and predictability of the features provided by the processor. From hardware side, the MERASA core is based on the SMT CarCore processor [6] which is binary compatible to the Infineon TriCore instruction set [3].

The MERASA system-level software [10] as a real-time operating system (RTOS) provides a fundament for applications running on the MERASA processor. The challenge is to guarantee an isolation of multiple hard real-time threads on memory and I/O resources or at least time-bounded access to avoid mutual and possibly unpredictable interferences. The intent of this isolation and bounding is also to enable an effective WCET analysis of application code. The resulting system software executes hard real-time threads in parallel on different cores of the multi-core MERASA processor. Moreover, it provides timing analysable synchronisation functions and memory management facilities for parallel applications. The hard real-time threads potentially run in concert with additional non real-time threads within the simultaneously multithreaded MERASA cores.

In this report we focus on the final MERASA system-level software which provides the fundamental building blocks of a real-time operating system (RTOS) on top of the MERASA multi-core processor and also integrates drivers to support pilot studies on the MERASA FPGA prototype.

This report is organised as follows: Section 2 gives an overview of requirements arising for a multi-core real-time operating system. In section 3 we present an architectural overview, followed by a short description of the user interface in section 4. Section 5 shows the implementation of the single parts developed to accomplish these requirements. Section 6 concludes this paper. The annex finally shows detailed informations on the API.

[1] **M**ulti-Core **E**xecution of Hard **R**eal-Time **A**pplications **S**upporting **A**nalysability. This research is funded by the European Community's Seventh Framework Programme under Grant Agreement number 216415. See www.merasa.org for details.

2. Requirements

In this section, we state the minimum requirements for a *Real-Time Operating System* (RTOS) for embedded systems with multithreaded and multi-core hardware. We show the basic properties that must be fulfilled in order to guarantee a correct execution of applications. Also, we regard requirements on the API.

2.1. Functional Requirements

In general, an operating system (OS) makes the usage of computer hardware possible. It provides an interface to access system resources like memory, I/O devices and to manage the execution of tasks. So we can summarise the common requirements:

- The OS has to manage processes and schedule processor time.
- Memory for the applications must be allocated and controlled.
- The OS must control and manage the connected devices.
- In case of errors and interrupts, they must be handled by the OS.

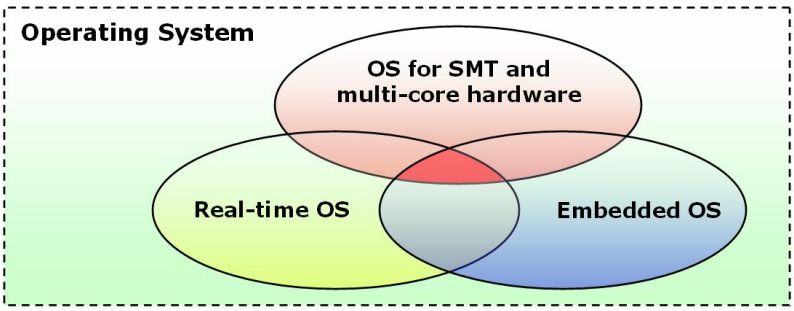


Figure 1: Combination of requirements for different OS types

Operating systems can be categorised into different classifications. So we can distinguish between single- and multi-user as well as single-threaded and multi-threaded systems. Depending on response time or execution mode, we can find real-time and non-real-time, embedded- or general-purpose computing operating systems. As our objective is the development of an RTOS for embedded systems with multithreaded and multi-core hardware, we need a combination of different fields. Figure 1 shows a symbolic intersection of the different fields of requirements and how they must be mixed together. So we first take a look at the concept of an RTOS and then add aspects regarding both embedded and SMT systems.

Concept of RTOSs

The key difference between general-purpose operating systems and real-time operating systems is the need for a deterministic timing behaviour. All operating system services must consume only known and expected amounts of time. No task is allowed to cause random delays and make an application miss real-time deadlines. So in most RTOSs tasks are scheduled according to a *priority based preemptive scheduling* scheme. Each task is assigned a priority, with higher values representing a need for earlier execution. The *preemptive* nature of the task scheduling enables a fast responsiveness. The scheduler is allowed to stop one task's execution, if another task needs to run immediately.

Regarding [2] we can summarise the general facets making an OS an RTOS:

- The RTOS has to be multi-threaded and preemptible.
- Either the notion of thread priority exists, or the RTOS provides a deadline driven scheduler.
- The RTOS supports predictable thread synchronisation mechanisms and especially avoids deadlocks caused by priority inversion.
- The timing behaviour of the RTOS must be known and predictable.

OSs in Embedded Environments

Embedded systems are mostly not recognisable as computers, instead they are hidden inside cars, aeroplanes or everyday objects surrounding and helping us in our live. A high connectivity to the environment through sensoric interfaces providing context data is typical.

The characteristic operation of embedded systems is limited by computer memory and processing power. The services they provide to their users are usually constrained by strict time deadlines. When using an OS in embedded environments, we also have to regard these restrictions on memory and performance.

So, we can outline also the requirements implicated from the field of embedded computing:

- The embedded OS must be very time and memory efficient.
- The OS has to be compact and concentrate on the most necessary functions.

OSs for Multithreaded and Multi-Core Hardware

Multithreading is the ability to concurrently run programs divided into subcomponents or threads on a single processor or within one processor core, whereas multi-core hardware offers real parallelism. However, both mechanisms promise a better utilisation of processors and other system resources. As a result they provide a scalable, modular environment upon which it is appropriate to write

application software. Working with several tasks in parallel, a multi-threaded or multi-core hardware can also cause a lot of new potential bugs to be introduced into an application. So we can add as specific requirement that an RTOS for multi-cores has to be aware of the following challenges:

- The OS must avoid race conditions and deadlocks caused by timing problems or commonly accessed resources like memory.
- Moreover, the OS has to provide synchronisation mechanisms, which enable an adjustment and communication between different processes running in parallel.

2.2. Requirements from Application Side

In order to enable an easy-to-handle application development, the interface of the operating system should be geared towards a common standard. As POSIX [7] is widely used also in the fields of embedded real-time systems (e.g. QNX [8]), it is useful to support functionalities following this standard. Thus, one can provide a familiar handling of parameters, return values and function names, e.g. when using synchronisation mechanisms for parallel applications. It will be easy to port applications developed for other systems using the POSIX interface as well (see sect. 4.5).

Based on these requirements concerning both functionalities and the user interface, we are now able to propose an architecture for the MERASA system level software that implements a combination of concepts concerning a RTOS for embedded environments with multithreaded and multi-core hardware.

3. Architectural overview

The design of the MERASA system-level software joins several well-known OS techniques. The basic kernel comprises the most important management functionalities following the microkernel principle. Additional functions may run outside this kernel as separate components.

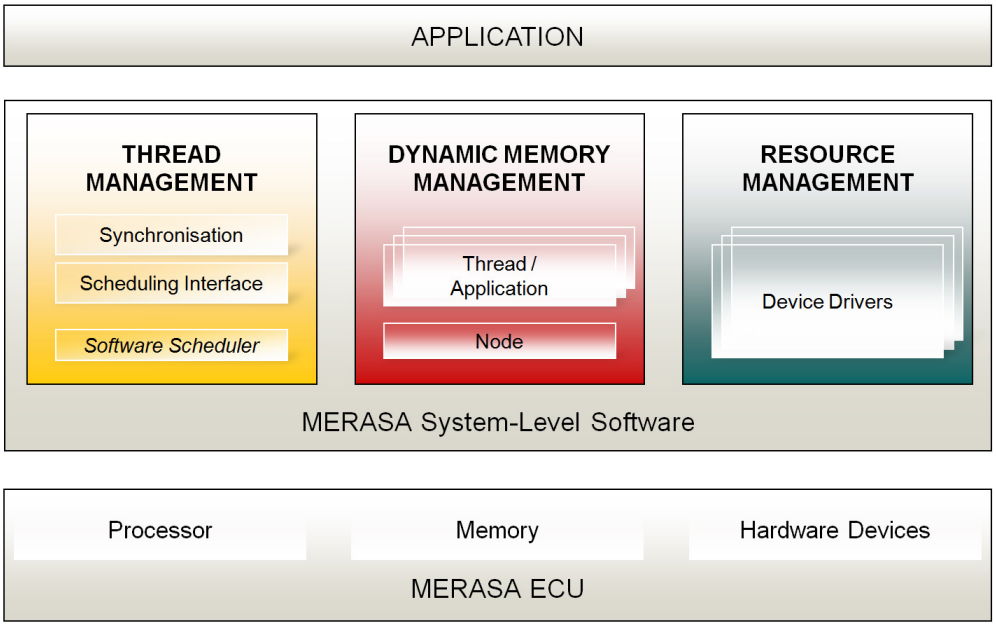


Figure 2: Architecture of the MERASA system-level software

Figure 2 gives an overview of the proposed architecture. The core of the MERASA system-level software contains three components: the Thread Management, the Dynamic Memory Management and the Resource Management. These three parts run on top of the MERASA hardware [9]. To give consideration to the real-time requirements the system-level software uses pre-allocation techniques. In an initial phase, all resources, which may potentially cause non real-time behaviour, are allocated. When the application starts running, real-time behaviour is guaranteed for all resource accesses. The next section will show in more detail how the application programming interface can be used. Section 5 describes the working of the kernel parts and especially how the real-time execution is ensured.

4. Application Programming Interface

In this section we describe the programming interface to access the MERASA system-level software (header files in the directory `include/`). This section is a short guide for application programmers to know where to find necessary functionalities, whereas the full specification is confined to the annex. In the first three parts of this section we specify the functions of the basic architectural components. The last part explains some common header files.

4.1. Thread Management

The Thread Management is implemented as a subset of POSIX functions:

pthread.h This header provides essential functions for the Thread Management. It allows the creation of threads and the management of the different hardware thread types (see sect. 5.1) and their scheduling parameters. Thread privileges are defined here as a base for a security manager. Moreover, it contains an interface to access the thread synchronisation mechanisms providing functions for mutex, conditional and barrier variables.

4.2. Dynamic Memory Management

For the usage of the Dynamic Memory Management it is necessary to utilise the definitions from the following header files:

memory.h This header represents the basic file of the memory management. It provides methods to allocate a specified amount of memory or to free previously allocated blocks of the current thread. As well, one can perform a copy of non-overlapping memory sections to another address.

memory-desc.h The Dynamic Memory Management of the MERASA system-level software supports the usage of different types of memory. This file provides functionalities to notify the Dynamic Memory Management about the availability of memory hardware.

4.3. Resource Management

One basic task of the MERASA system-level software is to enable the usage of computer hardware. To get a high level of flexibility, the hardware resources are accessed through device drivers. The interface to these drivers and the resource management are defined in the following files:

driver.h This header provides macros and data types to write an individual device driver for the MERASA system-level software. So it can be ensured that the compiled drivers have the correct file layout.

drivermanager.h This file enables the management of the device drivers. It contains functions to install and remove drivers from the system and for an access of the drivers' functions.

fcntl.h This header provides a function to open a device in the MERASA resource manager.

stropts.h Here, a function is included to control operations on a specific device.

unistd.h This file contains functions to read and write from devices.

A special resource currently included in the MERASA system-level software is the *Virtual Output*. It implements functions very similar to `stdio.h` with different format modifiers for various data types. In the MERASA simulator the output is written to `STDOUT`, prefaced with some special characters (%#). This enables an easy way to debug applications but will influence the timing behaviour of the application. For this reason, the virtual output also provides *fast-logging facilities* to output just few bytes of data. These functions come with less and predictable timing overhead. They require less than ten processor instructions.

log.h Especially for debugging purpose using the the virtual output this header gives several easy logging and fast-logging facilities. The output is thread safe and can be written to a log file or to `STDOUT`. There are five predefined loglevel-stages making it easy to distinguish between negligible debugging output, interesting warnings and important fatal errors. The level is set in the makefile (`Makefile_commons.mk`) for compilation.

vo.h This header defines the memory regions for the virtual output.

The final version of the MERASA system-level software also includes several device drivers to enable the usage of peripheral components of the FPGA prototype, like SPI, CAN, serial communication over UART, an LCD display, and a hardware timer to measure program execution times. Also, debugging utilities as e.g. buttons, LEDs and a seven-segment display on the FPGA prototype board can be used.

uart.h The UART interface of the MERASA FPGA prototype is used for communication by serial byte streams between the FPGA board and an external device. The UART device implements the RS232 protocol and provides baud rate adjustable by software. This file defines the device register map, providing symbolic constants to access the low-level hardware and the device driver API.

pio.h On the FPGA board we can use four userdefined push-button switches (sw4 to sw7), eight user-defined LEDs (D1 to D8) and a dual seven-segment display. All these components are a possibility to debug the system under development. They are connected to the MERASA processor over parallel input / output ports (PIO). The software components to control the mentioned ports can be found in this file.

lcd.h In order to get better debugging possibilities than the seven-segment dis-

play can deliver, we connected a 4x20 (4 lines, 20 chars per line) text LCD display to the FPGA board. An API to control this device can be found in this header.

timer.h To measure program execution times we implemented a hardware timer in our MERASA FPGA prototype. The timer is a 48-bit counter which is incremented in every cycle. Start, stop and read operations are called by software using the functions declared in timer.h.

spi.h The serial peripheral interface (SPI) of the MERASA FPGA prototype is an SPI master. It allows the FPGA board to communicate with an external SPI slave device over an SPI bus. This header defines the device register map, providing symbolic constants to access the low-level hardware, some macros to set or clear single status bits and the device driver API.

mcp2515.h The CAN interface of the MERASA FPGA prototype is an external module, which contains a stand-alone MCP2515 chip as controller area network (CAN) controller. This file defines the driver API and some useful macros for this interface.

4.4. Common header files

Besides the interface to the three main parts of the MERASA system-level software, there are several common header files:

config.h This file includes the general configuration of the MERASA system-level software. In many global definitions you can set the details, e.g. the number of used cores, the basic configuration of scheduling, the adresses of used thread memory etc.

error.h This header contains definitions of error numbers and error types. These are equal to the error number definitions defined by the POSIX standard.

sysmonitor.h Here, monitoring functionalities of system parameters are provided. It is possible to get detailed statistics on the usage of global and each thread's memory.

sys/types.h In this header platform-specific definitions of basic data types are defined.

merasa-ssw.h This file includes all MERASA system-level software headers.

4.5. POSIX interface

In compliance with the MERASA partners a subset of the POSIX interface was implemented, including basic functionalities for thread creation (with different attributes for scheduling) and thread join. Listing 1 shows how to use the interface from application side. Here, one hard real-time (HRT) thread executing function `f1` and one non real-time (NRT) thread executing `f2` is initialised. The

procedure `finish_thread_init()` signals the end of the initialisation phase to the system software, and the execution of all threads will start just then. Finally, the `join` function is called to suspend the calling thread and wait for both other threads to finish execution.

```
pthread_t my_hard_rt_thread;
pthread_t my_non_rt_thread;
pthread_attr_t my_hard_rt_attr;
pthread_attr_t my_non_rt_attr;

pthread_attr_setschedpolicy(&my_hard_rt_attr, SCHED_HRT);
pthread_create(&my_hard_rt_thread, &my_hard_rt_attr, &f1, NULL);

pthread_attr_setschedpolicy(&my_non_rt_attr, SCHED_NRT);
pthread_create(&my_non_rt_thread, &my_non_rt_attr, &f2, NULL);

finish_thread_init();

pthread_join(&my_hard_rt_thread);
pthread_join(&my_non_rt_thread);
```

Listing 1: Example for using the POSIX interface from application side

Moreover, also the synchronisation mechanisms are implemented following the POSIX standard. Thus, it is easy for application programmers to use the commonly known functionalities for mutex, conditional and barrier variables as we provide a familiar handling of parameters, return values and function names.

Besides the thread management, the resource management was enhanced covering a subset of the commonly used POSIX interface. We implemented basic operations like the generic `read` / `write` functions as well as driver-specific access by `open` / `close` and configuration by `ioctl` in a POSIX-compliant way.

5. Implementation

This section describes the implementation of the architectural parts of the final MERASA system-level software in more detail.

5.1. Thread Management

A basic goal of the system software is to provide a solid interface for an easy creation, suspension and wake-up of specific threads from application side. The MERASA system software achieves this goal by being compatible to the commonly used POSIX interface. The scheduling interface internally builds a correct and consistent baseline for the hardware by managing hard real-time (HRT) and non real-time (NRT) threads in a time-bounded fashion.

Moreover, it is necessary to fulfil the requirements on thread synchronisation. For this reason, the system software provides commonly used mechanisms like mutex, conditional and barrier variables. All these mechanisms are implemented following the POSIX standard. In the field of synchronisation, the challenges for a HRT execution on a multi-core processor are a time-predictable implementation of synchronisation primitives and a bounding of waiting times. This waiting time depends also on adhering to coding guidelines for the applications' programmer.

Scheduling Interface

The MERASA multi-core processor contains a two-level scheduler, which distributes threads over the different cores and over the different available thread slots of each core.

The processor provides the following interface to the system software for managing threads: Every HRT and NRT thread has its own Thread Control Block (TCB), which is an array of 256 bytes. In this array, the whole thread context, like register values, synchronisation and scheduling information is stored. All TCBs are located in segment `0x90000000` and aligned to 256 byte boundaries, i.e. TCB 1 starts at `0x90000100`, TCB 2 at `0x90000200`, etc.

TCB 0 is reserved for special data used by the hardware scheduler, it contains the heads of two double linked lists, one for the HRT threads (a 32 bit pointer at `0x90000014`) and one for the NRT threads (a 32 bit pointer at `0x90000018`). Each TCB contains two pointers `sched_next` and `sched_prev`, which are used for building these lists. On thread creation a TCB is put into one of the HRT or NRT lists using these pointers. The position of the TCB within a list indicates the order in which the corresponding threads are put into the cores' thread slots. Figure 3 shows an exemplary structure for such lists within the TCB memory. Concerning the timing behaviour, we benefit from the fact that the number of HRT threads is bounded by the number of cores, allowing only one HRT thread per core (to avoid interferences of HRT threads within a core).

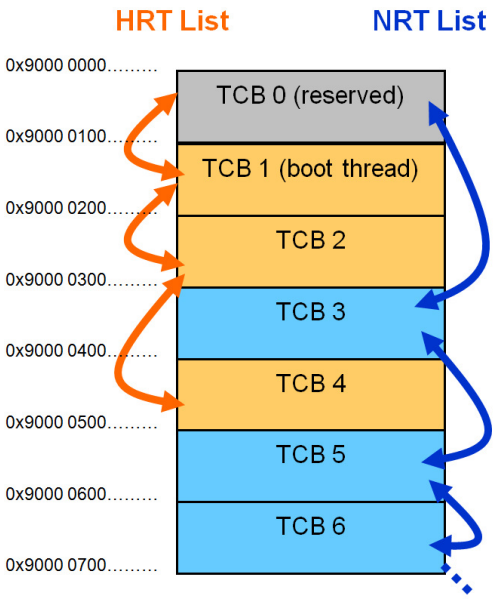


Figure 3: Two linked lists of TCBs, ready for the scheduler

As the process of thread creation cannot guarantee timing constraints, we decided to introduce an initial phase, where only one thread is running on one core. This boot thread can create new HRT and NRT threads only during this phase. The system software then fulfils all preparation jobs with non real-time behaviour, like memory allocation, to guarantee timing correctness during the following execution phase. From application side, it is very important to signal the end of the initialisation phase by calling exactly once the function `finish_thread_init()`. After this call no further creation of threads is possible, all needed threads must have been created during the initialisation phase.

The call to finish the initial phase and to start thread execution internally writes the heads of the HRT and NRT threads to the specific fields of the reserved TCB 0. This write access is snooped by the second level of the hardware scheduler, which in turn now starts operation. It begins traversing first the HRT list and puts each TCB onto another core, each in thread slot 0. This continues until the list is terminated or the maximum number of cores is reached. There is one special case: as slot 0 of core 0 is the boot thread (the only one active after a reset), the head of the HRT list must always point to TCB 1 (the boot thread) and then to the next HRT thread.

Writing the head pointer of the NRT list, which is also triggered by finishing the initial phase, also invokes the hardware scheduler: it starts traversing the NRT list. In detail it reads the first NRT thread's TCB from the specified memory location and writes it to core 0, thread slot 1. Following `sched_next` it puts the next threads into thread slot 1 of core 1 up to `MAX_CORES`. If the maximum number of cores is reached, it continues with thread slot 2 on each core, and so on. In figure 4, the distribution of the different TCBs over the slots is shown in

more detail for the list presented in figure 3.

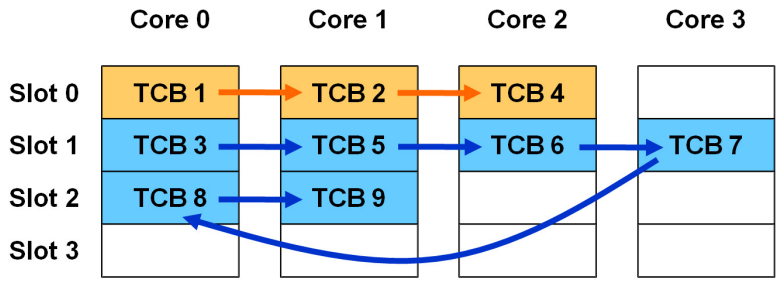


Figure 4: Distribution of the TCBs of figure 3 on a quad-core with four hardware thread slots per core

As already mentioned above, the developed hardware scheduler consists of two levels: a fixed priority (FP) scheduler in the issue stage of the pipeline and the initialisation logic to distribute threads over the slots in all cores. The FP scheduler works the following way: there are 4 slots per core and the issue logic chooses the thread with the highest priority that is ready. The HRT thread always has the highest priority within one core. The second level of the hardware scheduler is the initialisation logic, which provides an interface to tell the cores where to start program execution. It sets the program counters and register values for each thread slot according to the prepared TCB. In order to achieve higher flexibility of scheduling, it is possible to additionally use a software scheduler within each thread slot.

Synchronisation

The second basic part of the Thread Management of the MERASA system software besides the scheduling interface are the synchronisation mechanisms.

As base, a commonly known spinlock mechanism is implemented, which can be used to give only one thread access to a critical region. If another thread tries to access the same region, it performs *busy waiting* until the lock is free. Nevertheless, we can guarantee timing predictability, as the number of potentially waiting threads, i.e. the list of HRT threads is bounded by the number of cores in our architecture. However, the spinlock mechanism is only used in the system software for short critical regions. From architecture side, the implementation of spinlocks is based on the atomic swap instruction enabling good performance and predictable blocking times.

The spinlock methods are not provided to application programmers in order to advise the usage of mutex variables instead. The implementation is internally based on spinlocks, which cover only a very short critical region of a few cycles. Whenever a thread has to wait for a mutex to become free, it is suspended from execution and inserted to a waiting list connected to the specific mutex variable. As soon as the mutex is unlocked, the suspended threads are reactivated follow-

ing the waiting list. From a hard real-time point of view, it is essential to avoid common locks for HRT and NRT threads, because a HRT thread may have to wait for the release of a lock held by a NRT and therefore loses its property of being a HRT thread, i.e. its timing behaviour becomes unpredictable.

The thread management of the MERASA system software provides conditional variables as means of thread communication. By calling `wait` on a conditional variable, a thread is suspended and registered to a *waiting list*. This process is very similar to waiting on a locked mutex variable, as described above. As soon as the `signal` function is called, the first thread of the waiting list is unsuspended, on `broadcast` all waiting threads are unsuspended and continue execution.

Finally, in order to enable a simultaneous start of a specific section of different threads, the thread management provides barriers. In an initialisation function, the application programmer must set the number of threads which should wait at the barrier. Every time a thread calls the wait function, a counter, connected to the initialised barrier is incremented and the thread is suspended from execution. As soon as the incremented counter reaches the value which was set on initialisation, all threads in the waiting list get a signal to simultaneously continue execution.

Together with MERASA project partner Université Paul Sainrat Toulouse, we analysed the synchronisation techniques using the OTAWA WCET tool [1]. As a result, we can split the waiting times for threads into different components, dependent or independent of concurrently running threads. To summarise the outcome of the analysis, we could show that a WCET analyis of the synchronisation techniques is possible and its WCET can be bounded. A detailed explanation and further results can be found in [10].

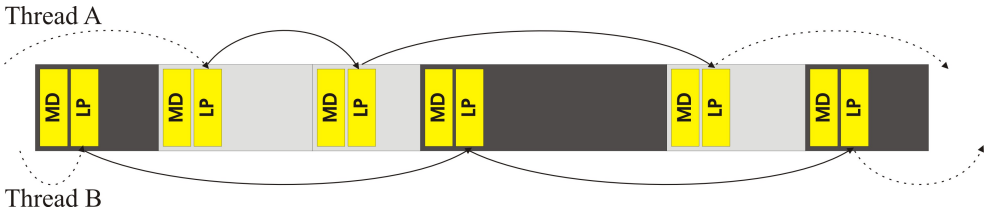
5.2. Dynamic Memory Management

In contrast to traditional non real-time memory management systems, it is necessary for our memory management to provide timing guarantees. We introduce a two-layered memory management and use memory pre-allocation. Thus, we minimise interferences of threads among each other and provide higher flexibility for applications at the same time.

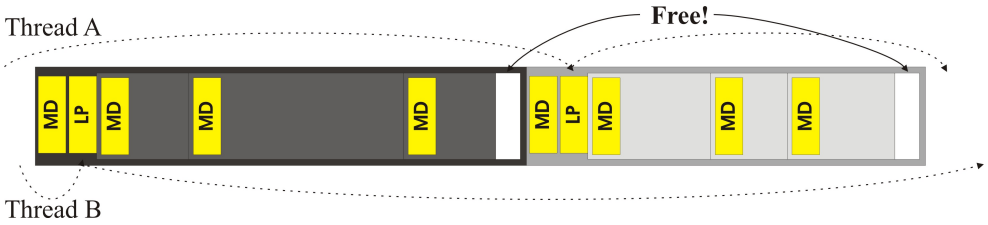
On the first layer – the *node* level – large blocks of memory are allocated. This allocation is performed in a mutually exclusive way to keep the state of memory consistent, so here a blocking of threads can occur. But as this is only done in the initialisation phase before threads are started, there are no influences on the system's timing behaviour during real-time operation. On the *thread* layer the memory management allocates memory to the program executed in the specific thread. This can be done without locking, because the memory is taken from the blocks pre-allocated in the node level exclusively for the thread.

Figure 5 shows another advantage of such a two-layered architecture. As it is always necessary to keep information, which memory block belongs to which

thread, management data is needed by putting them into a linked list including list pointers (LP). In contrast to a conventional (one-layered) allocation scheme, our list pointers need only be added to the large blocks on node level, as shown in 5(b). Thus, some memory can be saved and the management data needed to keep track of the owners is reduced.



(a) Conventional memory management; the blocks of the threads are highly mixed



(b) Two-stage memory management, the threads' memory is kept separated

Figure 5: Example layout of used memory with two threads; MD: Management data of the memory allocator, LP: List Pointers to keep track of thread's memory

Regarding the implementation, dynamic memory management on the node level is currently performed by an allocator based on Lea's allocator [4] (DLAlloc). On the thread level, the user can choose between various implementations of memory allocators. For non real-time applications, efficiency of memory usage can be achieved by a best-fit allocator like DLAlloc. This variant is space-conserving but hardly real-time capable. If a real-time application requires the flexibility of dynamic storage allocation, an allocator with bounded execution time can be used. The Two-Level Segregate Fit (TLSF) allocator [5] is a dynamic memory allocator specifically designed to meet real-time requirements. Using this alternative, the computation of worst-case execution time (WCET) is simplified. In DLAlloc, the execution time depends on the current state of the allocator and on the previous de-/allocations, whereas TLSF provides a bounded execution time regardless of its former operation at the cost of a higher internal fragmentation. The memory management supports different types of memory. It provides functionalities to define the configuration, i.e. beginning, end, length and the cost for the use. By this means, a high flexibility of memory structures is achieved and timing remains analysable.

However, for some parallel applications it is necessary to access an allocated memory region by multiple threads. As this is not possible in our two-level allocation mechanism due to the isolation of different threads' memory regions, we enhanced the MERASA system-level software by adding a mechanism to

split the whole heap into a private region and a shared region. In the private region of the heap the two-level allocation works as described above, whereas the shared region is reduced to one real-time capable allocation level. The region can also be regarded as one larger block allocated on node level allowing the access of multiple threads. The shared allocation during the applications' execution must be performed in a mutually exclusive way to keep the state of memory consistent. Nevertheless, we are able to guarantee timing predictability, as the number of potentially waiting threads, i.e. the list of HRT threads is bounded by the number of cores in our architecture.

5.3. Resource Management

It is a main task of the system software to enable the usage of computer hardware. As already mentioned, the MERASA system software follows the microkernel principles, i.e. manages only the most essential system resources like processing time and memory. To gain maximum flexibility, hardware devices are accessed through device drivers. These resources are managed by a dedicated *Resource Management* unit.

The implementation of the resource management, containing a driver manager, is a subset of the POSIX standard [7]. It contains the generic `open` / `close` operations as well as functions to access the device (`read` / `write` operations) and for configuration (`ioctl`). Valid configuration values and further parameters depend on the specific device and driver.

In the Resource Management the problem of concurrent use of devices can be reduced to thread synchronisation. For this, the Thread Manager already provides solutions. We assume that most operations can be dedicated to the non real-time initialisation phase and therefore have no influences on the timing behaviour of the execution phase. In general, there is no limitation on the number of drivers supported by the resource manager, however, the timing behaviour depends on this quantity. For the use in real-time applications, the number of threads sharing a resource must be limited to guarantee device accesses in bounded time. As the number of devices and threads an application uses is known in advance, constant-access-time handlers can be arranged during the preparation of the application's execution environment. Thus, device accesses can be performed in constant time, depending on their peripherals.

Software Drivers

The peripheral components of our FPGA prototype are a controller area network (CAN) interface, a serial peripheral interface (SPI), a universal asynchronous receiver/transmitter (UART) interface, a timer, a liquid crystal display (LCD) and a seven-segment display. Moreover, four input buttons and eight LEDs of the development board can be accessed. All these peripheral components are mainly used as communication interfaces to other devices, but also for debugging purposes. The software which should be executed on the board is loaded into

memory via UART.

The peripheral components of the MERASA FPGA prototype are memory mapped (to segment F). To access and control these devices, we integrated specific drivers, which are developed in C. More details on the peripheral components of the MERASA FPGA prototype as well as the usage of the specific drivers can be found in the annex.

6. Summary

In this report we presented the final MERASA system-level software as a fundament for applications running on the hard real-time capable multi-core MERASA processor. The architecture consists of three main parts: The *Thread Manager* provides a solid interface for an easy creation, suspension and wake-up of specific threads in a time-bounded fashion. It contains an interface to the two-level hardware scheduler of the MERASA processor and time-bounded mechanisms for thread synchronisation in order to support parallel applications. The *Dynamic Memory Management* eliminates interferences of different threads by providing a flexible two-layered memory management with memory pre-allocation and an isolation of threads' memory regions. The *Resource Management* enables the use of peripheral components and provides device drivers to support different pilot studies.

Regarding the different classes of requirements stated in section 2 we can summarise how they are fulfilled in detail: The MERASA system-level software is a solution for the RTOS support of parallel hard-real time applications on the new MERASA multi-core architecture. For thread synchronisation it provides time-bounded mutex, conditional and barrier variables. A real-time capable two-level scheduler is implemented in the MERASA hardware, so the system-level software guarantees a correct management of scheduling parameters supporting a known and predictable timing behaviour. All HRT operations on thread initialisation and memory pre-allocation that are not real-time capable are executed in an initialisation phase which is independent from the execution with timing guarantees. The system-level software is very compact because it concentrates on necessary functionalities and provides a fast and efficient way of execution. As the two-level dynamic memory management of the MERASA system-level software keeps the threads' memory separated on node level, the management effort of the memory blocks is reduced.

Moreover, our multi-core architecture is able to execute hard real-time threads in concert with additional non real-time threads. We ensure a complete isolation, so the hard real-time threads still meet their deadlines, while the processor also provides execution capacity to the non real-time threads.

A. Data Structure Documentation

A.1. cache_pipeline_t Struct Reference

`#include <cache-pipeline.h>`

Data Fields

- **uint16_t masks** [CACHE_PIPELINE_STAGES]
- **uint16_t stages**
- **pthread_barrier_t first_barrier**
- **pthread_barrier_t second_barrier**

A.1.1. Detailed Description

This struct represents the software pipeline.

Definition at line 59 of file cache-pipeline.h.

A.1.2. Field Documentation

A.1.2.1. uint16_t cache_pipeline_t::masks[CACHE_PIPELINE_STAGES]

masks for shared heap; one per core, i.e. only for HRT thread in each core

Definition at line 60 of file cache-pipeline.h.

A.1.2.2. uint16_t cache_pipeline_t::stages

stages of the pipeline

Definition at line 61 of file cache-pipeline.h.

A.1.2.3. pthread_barrier_t cache_pipeline_t::first_barrier

first barrier for clock_edge

Definition at line 62 of file cache-pipeline.h.

A.1.2.4. pthread_barrier_t cache_pipeline_t::second_barrier

second barrier for clock_edge

Definition at line 63 of file cache-pipeline.h.

The documentation for this struct was generated from the following file:

- **cache-pipeline.h**

A.2. CANMessage_t Struct Reference

`#include <mcp2515.h>`

Data Fields

- unsigned short **id**
- unsigned char **rtr**
- unsigned char **length**
- unsigned char **data** [8]

A.2.1. Detailed Description

Definition at line 47 of file mcp2515.h.

A.2.2. Field Documentation

A.2.2.1. unsigned short CANMessage_t::id

Definition at line 49 of file mcp2515.h.

A.2.2.2. unsigned char CANMessage_t::rtr

Definition at line 50 of file mcp2515.h.

A.2.2.3. unsigned char CANMessage_t::length

Definition at line 51 of file mcp2515.h.

A.2.2.4. unsigned char CANMessage_t::data[8]

Definition at line 52 of file mcp2515.h.

The documentation for this struct was generated from the following file:

- **mcp2515.h**

A.3. driver Struct Reference

`#include <driver.h>`

Data Fields

- void * **pi**
- const char * **name**
- **version_t version**
- **drv_init_fn_t init**
- **drv_cleanup_fn_t cleanup**
- **drvif_t * ops**
- **pthread_mutex_t lock**
- **thread_handler owner**
- struct **driver** * **sp_next**

A.3.1. Detailed Description

This struct describes a device **driver** (p. 24). Do not use it directly, instead publish your **driver** (p. 24) using the **SDRIVER** (p. 48) macro!

Definition at line 89 of file driver.h.

A.3.2. Field Documentation

A.3.2.1. void* driver::pi

Process image, only for internal use.

Definition at line 90 of file driver.h.

A.3.2.2. const char* driver::name

Name of the **driver** (p. 24).

Definition at line 91 of file driver.h.

A.3.2.3. version_t driver::version

Version of this **driver** (p. 24). This field is currently not used, it should be set to 0x10.

Definition at line 92 of file driver.h.

A.3.2.4. drv_init_fn_t driver::init

Initialisation function.

Definition at line 93 of file driver.h.

A.3.2.5. drv_cleanup_fn_t driver::cleanup

Cleanup function; currently unused.

Definition at line 94 of file driver.h.

A.3.2.6. drvif_t* driver::ops

Operations struct.

Definition at line 95 of file driver.h.

A.3.2.7. pthread_mutex_t driver::lock

Lock for this **driver** (p. 24).

Definition at line 96 of file driver.h.

A.3.2.8. thread_handler driver::owner

Current owner of this **driver** (p. 24), -1 if unused.

Definition at line 97 of file driver.h.

A.3.2.9. struct driver* driver::sp_next [read]

Pointer for a thread's **driver** (p. 24) stack.

Definition at line 98 of file driver.h.

The documentation for this struct was generated from the following file:

- **driver.h**

A.4. driver_interface Struct Reference

`#include <driver.h>`

Data Fields

- **drv_open_fn open**
- **drv_close_fn close**
- **drv_read_fn read**
- **drv_write_fn write**
- **drv_ioctl_fn ioctl**

A.4.1. Detailed Description

This struct holds the functions a **driver** (p. 24) must implement, i.e. open, close, read, write and ioctl.

Definition at line 80 of file driver.h.

A.4.2. Field Documentation

A.4.2.1. drv_open_fn driver_interface::open

Definition at line 81 of file driver.h.

A.4.2.2. drv_close_fn driver_interface::close

Definition at line 82 of file driver.h.

A.4.2.3. drv_read_fn driver_interface::read

Definition at line 83 of file driver.h.

A.4.2.4. drv_write_fn driver_interface::write

Definition at line 84 of file driver.h.

A.4.2.5. drv_ioctl_fn driver_interface::ioctl

Definition at line 85 of file driver.h.

The documentation for this struct was generated from the following file:

- **driver.h**

A.5. mem_cfg_data Struct Reference

`#include <types.h>`

Data Fields

- char * **begin**
- char * **end**
- size_t **length**
- **uint32_t access_cycles**
- **int32_t cost**
- **uint32_t flags**
- char * **brk**
- void * **binlist**
- **pthread_mutex_t mutex**

A.5.1. Detailed Description

This struct is for the definition of a memory area.

Definition at line 160 of file types.h.

A.5.2. Field Documentation

A.5.2.1. char* mem_cfg_data::begin

first byte of memory area

Definition at line 161 of file types.h.

A.5.2.2. char* mem_cfg_data::end

first byte after the memory area

Definition at line 162 of file types.h.

A.5.2.3. size_t mem_cfg_data::length

length of memory area

Definition at line 163 of file types.h.

A.5.2.4. uint32_t mem_cfg_data::access_cycles

Average number of cycles to accesss one word of this memory

Definition at line 164 of file types.h.

A.5.2.5. int32_t mem_cfg_data::cost

cost for use of this memory

Definition at line 165 of file types.h.

A.5.2.6. uint32_t mem_cfg_data::flags

flags for further use

Definition at line 166 of file types.h.

A.5.2.7. char* mem_cfg_data::brk

current break (allocated up to this address)

Definition at line 167 of file types.h.

A.5.2.8. void* mem_cfg_data::binlist

the gmalloc bins for this memory; usually located at the start of the memory

Definition at line 168 of file types.h.

A.5.2.9. pthread_mutex_t mem_cfg_data::mutex

mutex for gmalloc

Definition at line 169 of file types.h.

The documentation for this struct was generated from the following file:

- **types.h**

A.6. memorystatistics Struct Reference

`#include <sysmonitor.h>`

Data Fields

- size_t **used**
- **uint32_t freepages_count**
- size_t **freepages_size**
- size_t **min_freepagesize**
- size_t **max_freepagesize**
- size_t **max_ext**

A.6.1. Detailed Description

describes the state of global memory

Definition at line 58 of file sysmonitor.h.

A.6.2. Field Documentation

A.6.2.1. size_t memorystatistics::used

total inuse

Definition at line 59 of file sysmonitor.h.

A.6.2.2. uint32_t memorystatistics::freepages_count

the count of free pages

Definition at line 60 of file sysmonitor.h.

A.6.2.3. size_t memorystatistics::freepages_size

size of all free pages

Definition at line 61 of file sysmonitor.h.

A.6.2.4. size_t memorystatistics::min_freepagesize

size of greatest free page

Definition at line 62 of file sysmonitor.h.

A.6.2.5. size_t memorystatistics::max_freepagesize

size of smallest free page

Definition at line 63 of file sysmonitor.h.

A.6.2.6. size_t memorystatistics::max_ext

maximum amount of memory that can be allocated using sbrk()

Definition at line 64 of file sysmonitor.h.

The documentation for this struct was generated from the following file:

- **sysmonitor.h**

A.7. pthread_attr_t Struct Reference

`#include <types.h>`

Data Fields

- **sched_param param**
- **memory_t * mem**
- **uint32_t flags**
- **uint32_t heapsize**
- enum **sched_policy policy**

A.7.1. Detailed Description

POSIX thread attributes

Definition at line 178 of file types.h.

A.7.2. Field Documentation

A.7.2.1. sched_param pthread_attr_t::param

Definition at line 179 of file types.h.

A.7.2.2. memory_t* pthread_attr_t::mem

Definition at line 180 of file types.h.

A.7.2.3. uint32_t pthread_attr_t::flags

Definition at line 181 of file types.h.

A.7.2.4. uint32_t pthread_attr_t::heapsize

Definition at line 182 of file types.h.

A.7.2.5. enum sched_policy pthread_attr_t::policy

Definition at line 183 of file types.h.

The documentation for this struct was generated from the following file:

- **types.h**

A.8. pthread_barrier_t Struct Reference

`#include <types.h>`

Data Fields

- int **needed**
- int **called**
- **pthread_mutex_t mutex**
- **pthread_cond_t cond**

A.8.1. Detailed Description

Barrier - *not fully tested yet*

Definition at line 145 of file types.h.

A.8.2. Field Documentation

A.8.2.1. int pthread_barrier_t::needed

Definition at line 146 of file types.h.

A.8.2.2. int pthread_barrier_t::called

Definition at line 147 of file types.h.

A.8.2.3. pthread_mutex_t pthread_barrier_t::mutex

Definition at line 148 of file types.h.

A.8.2.4. pthread_cond_t pthread_barrier_t::cond

Definition at line 149 of file types.h.

The documentation for this struct was generated from the following file:

- **types.h**

A.9. pthread_barrierattr_t Struct Reference

`#include <types.h>`

Data Fields

- **uint32_t test**

A.9.1. Detailed Description

Barrier attributes

Definition at line 152 of file types.h.

A.9.2. Field Documentation

A.9.2.1. uint32_t pthread_barrierattr_t::test

Definition at line 153 of file types.h.

The documentation for this struct was generated from the following file:

- **types.h**

A.10. pthread_cond Struct Reference

`#include <types.h>`

Data Fields

- **pthread_mutex_t * mutex**
- **tcb_t * waitlist_first_out**
- **tcb_t * waitlist_last_out_hrt**
- **tcb_t * waitlist_last_out**

A.10.1. Detailed Description

Conditional variable

Definition at line 134 of file types.h.

A.10.2. Field Documentation

A.10.2.1. pthread_mutex_t* pthread_cond::mutex

Definition at line 135 of file types.h.

A.10.2.2. tcb_t* pthread_cond::waitlist_first_out

Definition at line 136 of file types.h.

A.10.2.3. tcb_t* pthread_cond::waitlist_last_out_hrt

Definition at line 137 of file types.h.

A.10.2.4. tcb_t* pthread_cond::waitlist_last_out

Definition at line 138 of file types.h.

The documentation for this struct was generated from the following file:

- **types.h**

A.11. pthread_condattr_t Struct Reference

`#include <types.h>`

Data Fields

- **uint32_t test**

A.11.1. Detailed Description

Conditional attributes

Definition at line 141 of file types.h.

A.11.2. Field Documentation

A.11.2.1. uint32_t pthread_condattr_t::test

Definition at line 142 of file types.h.

The documentation for this struct was generated from the following file:

- **types.h**

A.12. pthread_mutex Struct Reference

`#include <types.h>`

Data Fields

- **cc_spinlock_t the_lock**
- **cc_spinlock_t guard**
- **thread_handler owner**
- **sched_t prev_sched**
- **tcb_t * waitlist_first_out**
- **tcb_t * waitlist_last_out_hrt**
- **tcb_t * waitlist_last_out**

A.12.1. Detailed Description

Mutex variable

Definition at line 120 of file types.h.

A.12.2. Field Documentation

A.12.2.1. cc_spinlock_t pthread_mutex::the_lock

Definition at line 121 of file types.h.

A.12.2.2. cc_spinlock_t pthread_mutex::guard

Definition at line 122 of file types.h.

A.12.2.3. thread_handler pthread_mutex::owner

Definition at line 123 of file types.h.

A.12.2.4. sched_t pthread_mutex::prev_sched

Scheduling parameters before the guarded critical block

Definition at line 124 of file types.h.

A.12.2.5. tcb_t* pthread_mutex::waitlist_first_out

Definition at line 125 of file types.h.

A.12.2.6. tcb_t* pthread_mutex::waitlist_last_out_hrt

Definition at line 126 of file types.h.

A.12.2.7. tcb_t* pthread_mutex::waitlist_last_out

Definition at line 127 of file types.h.

The documentation for this struct was generated from the following file:

- **types.h**

A.13. pthread_mutexattr_t Struct Reference

`#include <types.h>`

Data Fields

- **uint32_t test**

A.13.1. Detailed Description

Mutex attributes

Definition at line 130 of file types.h.

A.13.2. Field Documentation

A.13.2.1. uint32_t pthread_mutexattr_t::test

Definition at line 131 of file types.h.

The documentation for this struct was generated from the following file:

- **types.h**

A.14. sched_param Struct Reference

`#include <types.h>`

Data Fields

- **sched_t sched_priority**

A.14.1. Detailed Description

Scheduling parameter

Definition at line 110 of file types.h.

A.14.2. Field Documentation

A.14.2.1. sched_t sched_param::sched_priority

Definition at line 111 of file types.h.

The documentation for this struct was generated from the following file:

- **types.h**

A.15. thread_memorystatistics Struct Reference

`#include <sysmonitor.h>`

Data Fields

- size_t **reserved**
- size_t **used**
- **uint32_t freechunks_count**
- size_t **min_freechunksize**
- size_t **max_freechunksize**

A.15.1. Detailed Description

describes the state of a thread's memory

Definition at line 75 of file sysmonitor.h.

A.15.2. Field Documentation

A.15.2.1. size_t thread_memorystatistics::reserved

Definition at line 76 of file sysmonitor.h.

A.15.2.2. size_t thread_memorystatistics::used

Definition at line 77 of file sysmonitor.h.

A.15.2.3. uint32_t thread_memorystatistics::freechunks_count

Definition at line 78 of file sysmonitor.h.

A.15.2.4. size_t thread_memorystatistics::min_freechunksize

Definition at line 79 of file sysmonitor.h.

A.15.2.5. size_t thread_memorystatistics::max_freechunksize

Definition at line 80 of file sysmonitor.h.

The documentation for this struct was generated from the following file:

- **sysmonitor.h**

B. File Documentation

B.1. cache-pipeline.h File Reference

Functionalities for a software pipeline.

`#include <sys/types.h>`

`#include <config.h>`

Data Structures

- struct **cache_pipeline_t**

Defines

- #define **PIPELINE_H_** 1
- #define **CORE_REGISTER_GLOBAL** 0xb500
- #define **CORE_REGISTER_SHARED** 0xb504
- #define **CORE_REGISTER_PRIVATE** 0xb508

Enumerations

- enum **cache_type** { **CACHE_GLOBAL** = 0, **CACHE_SHARED**, **CACHE_PRIVATE** }

Functions

- void **set_bank_assignment** (enum **cache_type** type, **int16_t** mask)
- void **__init_cache_pipeline** (**int16_t** stages, **int32_t** bytes_of_cache)
- void **clock_edge** (**int16_t** stage)
- void **start_pipeline** (**int16_t** stage)
- void **stop_pipeline** (**int16_t** stage)

B.1.1. Detailed Description

The MERASA processor introduces bankization, a cache partition technique in which the cache is partitioned into banks, giving to each thread a subset of the total number of banks that no other thread can use. Moreover, bankization allows to remapping at runtime the banks assigned to the different threads such that the data seen by one thread can be accessed by another without requiring to move data among banks. This file provides functionalities to enable software pipelining in order to switch cache banks between different threads.

Definition in file **cache-pipeline.h**.

B.1.2. Define Documentation

B.1.2.1. #define PIPELINE_H_ 1

Definition at line 40 of file cache-pipeline.h.

B.1.2.2. #define CORE_REGISTER_GLOBAL 0xb500

Core register for global mask.

Definition at line 50 of file cache-pipeline.h.

B.1.2.3. #define CORE_REGISTER_SHARED 0xb504

Core register for shared mask.

Definition at line 52 of file cache-pipeline.h.

B.1.2.4. #define CORE_REGISTER_PRIVATE 0xb508

Core register for private mask.

Definition at line 54 of file cache-pipeline.h.

B.1.3. Enumeration Type Documentation

B.1.3.1. enum cache_type

Enumerator:

CACHE_GLOBAL
CACHE_SHARED
CACHE_PRIVATE

Definition at line 66 of file cache-pipeline.h.

B.1.4. Function Documentation

B.1.4.1. void set_bank_assignment (enum cache_type *type*, int16_t *mask*)

Sets a cache mask to the specific core register (global, shared, private). Possible values for parameter type are:

- `CACHE_GLOBAL`
- `CACHE_SHARED`
- `CACHE_PRIVATE`

B.1.4.2. void __init_cache_pipeline (int16_t *stages*, int32_t *bytes_of_cache*)

Initialises a cache pipeline with a specified number of stages and cache size.

It sets the number of stages and assigns to the first stage/thread of the pipelined parallel application its corresponding set of banks based on the cache size given to each stage. The size is forced to be a multiple of the size of a cache bank.

B.1.4.3. void clock_edge (int16_t *stage*)

This function is called to signal that the work of a round is done.

It is the core of the bankization technique which ensures the correct functionality of the bankization technique by synchronizing the start of all threads. To do so, it uses two mutex: first_barrier and second_barrier. The former ensures that all threads have finished its computation. The latter ensures that the new bank assignment has been done completely. The first stage is the one in charge of generating the banks reassignations.

B.1.4.4. void start_pipeline (int16_t *stage*)

This function starts the pipeline.

It stalls each thread (identified by the input parameter stage) until the data coming from the previous thread is available. This function is composed of a loop controlled by the number of stages that remain to execute the thread stage for the first time. At every iteration, the clock edge function is called.

B.1.4.5. void stop_pipeline (int16_t *stage*)

This function stops the pipeline.

This function does the same as start_pipeline but at the end of the execution. Thus, it stalls each thread (identified by the input parameter stage) until all the subsequent threads finish.

B.2. config.h File Reference

Global definitions for general configuration.

```
#include <sys/types.h>
```

Defines

- #define **NO_CORES** 4
- #define **NRT_SLOTS_PER_CORE** 3
- #define **HRT_THREADS** NO_CORES
- #define **NRT_THREADS** (NO_CORES * NRT_SLOTS_PER_CORE)
- #define **__USTACK_SIZE** 16384
- #define **HRT_STACK_BASE** ((**address**) 0xb0000000)
- #define **CSA_BASE** ((**address**) 0xd0000000)
- #define **HRT_CSA_CNT** 64
- #define **NRT_CSA_CNT** 64
- #define **LOGBUFFLEN** 512
- #define **CACHE_PIPELINE_STAGES** 4
- #define **BYTES_OF_CACHE** 4096
- #define **TCB_BEGIN** 0x90000100
- #define **MSS_HRT** (*((**uint32_t** volatile *) 0x90000014))
- #define **MSS_NRT** (*((**uint32_t** volatile *) 0x90000018))
- #define **MSS_MY_CORE** (*((**uint32_t** volatile *) 0x90000000))
- #define **MSS_MY_SLOT** (*((**uint32_t** volatile *) 0x90000004))
- #define **MSS_MY_TCB** (*((**uint32_t** volatile *) 0x90000008))
- #define **STM_TIM0** (*((**uint32_t** volatile *) 0xF0000210))

B.2.1. Detailed Description

This header file includes the general configuration of the MERASA System-Level Software. In many global definitions you can set the details, e.g. the number of used cores, the basic configuration of scheduling, the adresses of used thread memory etc.

Definition in file **config.h**.

B.2.2. Define Documentation

B.2.2.1. #define NO_CORES 4

The number of processor cores used in the System Software. Of course, the distribution of threads over the cores is always done by the scheduler, but this

definition makes sure from software side, that there will not be more threads created than the processor can handle.

Definition at line 54 of file config.h.

B.2.2.2. #define NRT_SLOTS_PER_CORE 3

The number of non real-time slots per core. Usually, one MERASA core provides four slots - the first is for hard real-time and the three others for non real-time threads.

Definition at line 57 of file config.h.

B.2.2.3. #define HRT_THREADS NO_CORES

The number of hard real-time threads. Usually, there is one hard real-time thread per core.

Definition at line 63 of file config.h.

B.2.2.4. #define NRT_THREADS (NO_CORES * NRT_SLOTS_PER_CORE)

The number of non real-time threads. Usually, there is one hard-real time thread per core, all others are non real-time threads.

Definition at line 66 of file config.h.

B.2.2.5. #define __USTACK_SIZE 16384

The size of the reserved user stack.

Definition at line 69 of file config.h.

B.2.2.6. #define HRT_STACK_BASE ((address) 0xb0000000)

The base address of the common stack for HRT threads (optimised for the use of a scratchpad).

Definition at line 72 of file config.h.

B.2.2.7. #define CSA_BASE ((address) 0xd0000000)

The size of one context save area (CSA). This is necessary to save the context (i.e. register values ...) of a function before performing another call. The base address of the reserved context save areas (CSA).

Definition at line 78 of file config.h.

B.2.2.8. #define HRT_CSA_CNT 64

The number of context save areas reserved for hard real-time threads.

Definition at line 81 of file config.h.

B.2.2.9. #define NRT_CSA_CNT 64

The number of context save areas reserved for non real-time threads.

Definition at line 84 of file config.h.

B.2.2.10. #define LOGBUFFLEN 512

The buffer length for log information in the virtual output.

Definition at line 90 of file config.h.

B.2.2.11. #define CACHE_PIPELINE_STAGES 4

Initialisation of the Cache Pipeline, number of pipeline stages.

Definition at line 93 of file config.h.

B.2.2.12. #define BYTES_OF_CACHE 4096

Initialisation of the Cache Pipeline, total size of cache in bytes

Definition at line 96 of file config.h.

B.2.2.13. #define TCB_BEGIN 0x90000100

The maximum number of threads. This is only a theoretical value, because the real number of threads is determined by the core and the available slots per core. The base address of Thread Control Blocks (TCBs). The range before TCB_-BEGIN is reserved for some specific scheduling information (see the MSS_∗ definitions for details). The begin address for the first Thread Control Block (TCB). This adress is usually used for the boot thread.

Definition at line 110 of file config.h.

B.2.2.14. #define MSS_HRT (∗((uint32_t volatile ∗) 0x90000014))

The end address of the Thread Control Blocks (first byte after). A pointer to the address of the first TCB in the list of hard real-time threads.

Definition at line 116 of file config.h.

B.2.2.15. #define MSS_NRT (∗((uint32_t volatile ∗) 0x90000018))

A pointer to the address of the first TCB in the list of non real-time threads.

Definition at line 119 of file config.h.

B.2.2.16. #define MSS_MY_CORE (*((uint32_t volatile *) 0x90000000))

Contains the core number of the actual thread.

Definition at line 122 of file config.h.

Referenced by get_current_core().

B.2.2.17. #define MSS_MY_SLOT (*((uint32_t volatile *) 0x90000004))

Contains the slot number of the actual thread

Definition at line 125 of file config.h.

Referenced by get_current_slot().

B.2.2.18. #define MSS_MY_TCB (*((uint32_t volatile *) 0x90000008))

Contains the TCB address of the actual thread

Definition at line 128 of file config.h.

B.2.2.19. #define STM_TIM0 (*((uint32_t volatile *) 0xF0000210))

Contains the actual system time.

Definition at line 131 of file config.h.

B.3. driver.h File Reference

Macros for writing MERASA device drivers.

`#include <pthread.h>`

`#include <stdarg.h>`

`#include <sys/types.h>`

Data Structures

- struct **driver_interface**
- struct **driver**

Defines

- #define **SDRIVER**(n, v, i, c, o)

Typedefs

- typedef **int32_t**(∗ **drv_init_fn_t**)(const void ∗)
- typedef **int32_t**(∗ **drv_cleanup_fn_t**)(void)
- typedef **int32_t**(∗ **drv_open_fn**)(void)
- typedef **int32_t**(∗ **drv_close_fn**)(void)
- typedef size_t(∗ **drv_read_fn**)(void ∗, size_t)
- typedef size_t(∗ **drv_write_fn**)(const void ∗, size_t)
- typedef size_t(∗ **drv_ioctl_fn**)(**uint32_t**, va_list)
- typedef struct **driver_interface drvif_t**
- typedef struct **driver driver_t**

B.3.1. Detailed Description

This file provides macros and data types that are necessary to write a hardware device **driver** (p. 24) for the MERASA operating system. For examples how to use them, see existing **driver** (p. 24) files. Please make sure to include all necessary functions within the **driver** (p. 24) program. The DriverManager will only resolve dependencies to OS API calls provided in the include directory!

Definition in file **driver.h**.

B.3.2. Define Documentation

B.3.2.1. #define SDRIVER(n, v, i, c, o)

Value:

```
struct driver sdriver_ ##n = { \
        NULL, \
        #n, \
        v, \
        i, \
        c, \
        o, \
        {}, \
        NO_THREAD, \
        NULL }
```

Macro for a static **driver** (p. 24).

Use this macro to publish your own drivers. For examples how to use them, see existing **driver** (p. 24) files.

Parameters:

- ***n*** Name of the **driver** (p. 24).
- ***v*** Version of the **driver** (p. 24) (shoud be set to 0x10).
- ***i*** Initialisation function.
- ***c*** Cleanup function.
- ***o*** Operations struct (**drvif_t** (p. 50)).

Definition at line 115 of file driver.h.

B.3.3. Typedef Documentation

B.3.3.1. typedef int32_t(* drv_init_fn_t)(const void *)

The initialisation function of a **driver** (p. 24). The passed pointer points to the device's start address in memory. Currently, this function must not fail!

Definition at line 58 of file driver.h.

B.3.3.2. typedef int32_t(* drv_cleanup_fn_t)(void)

The cleanup function of a **driver** (p. 24).

Definition at line 61 of file driver.h.

B.3.3.3. typedef int32_t(* drv_open_fn)(void)

The open function of a **driver** (p. 24).

Definition at line 64 of file driver.h.

B.3.3.4. typedef int32_t(* drv_close_fn)(void)

The close function of a **driver** (p. 24).

Definition at line 67 of file driver.h.

B.3.3.5. typedef size_t(* drv_read_fn)(void *, size_t)

The read function of a **driver** (p. 24).

Definition at line 70 of file driver.h.

B.3.3.6. typedef size_t(* drv_write_fn)(const void *, size_t)

The write function of a **driver** (p. 24).

Definition at line 73 of file driver.h.

B.3.3.7. typedef size_t(* drv_ioctl_fn)(uint32_t, va_list)

The ioctl function of a **driver** (p. 24).

Definition at line 76 of file driver.h.

B.3.3.8. typedef struct driver_interface drvif_t

B.3.3.9. typedef struct driver driver_t

B.4. drivermanager.h File Reference

Macros to declare and register a **driver** (p. 24).

Defines

- #define **EXTERN_DRIVER**(name) extern **driver_t** sdriver_-##name;
- #define **REGISTER_DRIVER**(name) &sdriver_##name

B.4.1. Detailed Description

This file provides macros necessary to declare and register a **driver** (p. 24) in the user application.

Definition in file **drivermanager.h**.

B.4.2. Define Documentation

B.4.2.1. #define EXTERN_DRIVER(name) extern driver_t sdriver_-##name;

Simple macro for the declaration of a **driver** (p. 24).

Definition at line 59 of file drivermanager.h.

B.4.2.2. #define REGISTER_DRIVER(name) &sdriver_##name

Simple macro to register a **driver** (p. 24) in the user application.

Definition at line 65 of file drivermanager.h.

B.5. error.h File Reference

Definitions of error numbers.

```
#include <sys/types.h>
```

Defines

- #define **E_OK** 0
- #define **EPERM** 1
- #define **ENOENT** 2
- #define **ESRCH** 3
- #define **EINTR** 4
- #define **EIO** 5
- #define **ENXIO** 6
- #define **E2BIG** 7
- #define **ENOEXEC** 8
- #define **EBADF** 9
- #define **ECHILD** 10
- #define **EAGAIN** 11
- #define **ENOMEM** 12
- #define **EACCES** 13
- #define **EFAULT** 14
- #define **ENOTBLK** 15
- #define **EBUSY** 16
- #define **EEXIST** 17
- #define **EXDEV** 18
- #define **ENODEV** 19
- #define **ENOTDIR** 20
- #define **EISDIR** 21
- #define **EINVAL** 22
- #define **ENFILE** 23
- #define **EMFILE** 24
- #define **ENOTTY** 25
- #define **ETXTBSY** 26
- #define **EFBIG** 27
- #define **ENOSPC** 28
- #define **ESPIPE** 29
- #define **EROFS** 30
- #define **EMLINK** 31

- #define **EPIPE** 32
- #define **EDOM** 33
- #define **ERANGE** 34

Functions

- **error_t get_errno** (void)
- void **set_errno** (**error_t** errno)

B.5.1. Detailed Description

This file contains definitions of error numbers, similar to POSIX.

Definition in file **error.h**.

B.5.2. Define Documentation

B.5.2.1. #define E_OK 0

Definition at line 53 of file error.h.

B.5.2.2. #define EPERM 1

Operation not permitted

Definition at line 57 of file error.h.

B.5.2.3. #define ENOENT 2

No such file or directory

Definition at line 58 of file error.h.

B.5.2.4. #define ESRCH 3

No such process

Definition at line 59 of file error.h.

B.5.2.5. #define EINTR 4

Interrupted system call

Definition at line 60 of file error.h.

B.5.2.6. #define EIO 5

I/O error

Definition at line 61 of file error.h.

B.5.2.7. #define ENXIO 6

No such device or address

Definition at line 62 of file error.h.

B.5.2.8. #define E2BIG 7

Argument list too long

Definition at line 63 of file error.h.

B.5.2.9. #define ENOEXEC 8

Exec format error

Definition at line 64 of file error.h.

B.5.2.10. #define EBADF 9

Bad file number

Definition at line 65 of file error.h.

B.5.2.11. #define ECHILD 10

No child processes

Definition at line 66 of file error.h.

B.5.2.12. #define EAGAIN 11

Try again

Definition at line 67 of file error.h.

B.5.2.13. #define ENOMEM 12

Out of memory

Definition at line 68 of file error.h.

B.5.2.14. #define EACCES 13

Permission denied

Definition at line 69 of file error.h.

B.5.2.15. #define EFAULT 14

Bad address

Definition at line 70 of file error.h.

B.5.2.16. #define ENOTBLK 15

Block device required

Definition at line 71 of file error.h.

B.5.2.17. #define EBUSY 16

Device or resource busy

Definition at line 72 of file error.h.

B.5.2.18. #define EEXIST 17

File exists

Definition at line 73 of file error.h.

B.5.2.19. #define EXDEV 18

Cross-device link

Definition at line 74 of file error.h.

B.5.2.20. #define ENODEV 19

No such device

Definition at line 75 of file error.h.

B.5.2.21. #define ENOTDIR 20

Not a directory

Definition at line 76 of file error.h.

B.5.2.22. #define EISDIR 21

Is a directory

Definition at line 77 of file error.h.

B.5.2.23. #define EINVAL 22

Invalid argument

Definition at line 78 of file error.h.

B.5.2.24. #define ENFILE 23

File table overflow

Definition at line 79 of file error.h.

B.5.2.25. #define EMFILE 24

Too many open files

Definition at line 80 of file error.h.

B.5.2.26. #define ENOTTY 25

Not a typewriter

Definition at line 81 of file error.h.

B.5.2.27. #define ETXTBSY 26

Text file busy

Definition at line 82 of file error.h.

B.5.2.28. #define EFBIG 27

File too large

Definition at line 83 of file error.h.

B.5.2.29. #define ENOSPC 28

No space left on device

Definition at line 84 of file error.h.

B.5.2.30. #define ESPIPE 29

Illegal seek

Definition at line 85 of file error.h.

B.5.2.31. #define EROFS 30

Read-only file system

Definition at line 86 of file error.h.

B.5.2.32. #define EMLINK 31

Too many links

Definition at line 87 of file error.h.

B.5.2.33. #define EPIPE 32

Broken pipe

Definition at line 88 of file error.h.

B.5.2.34. #define EDOM 33

Math argument out of domain of func

Definition at line 89 of file error.h.

B.5.2.35. #define ERANGE 34

Math result not representable

Definition at line 90 of file error.h.

B.5.3. Function Documentation

B.5.3.1. error_t get_errno (void)

Get the error number of the actual thread.

Returns:

E_OK if there is no error, otherwise the specified error number.

B.5.3.2. void set_errno (error_t *errno*)

Set the error number of the actual thread.

B.6. fcntl.h File Reference

Open or create a device for reading or writing.

Functions

- int **open** (const char *path, int flags,...)

B.6.1. Detailed Description

Definition in file **fcntl.h**.

B.6.2. Function Documentation

B.6.2.1. int open (const char * *path*, int *flags*, ...)

Open or create a device for reading or writing.

The device name specified by path is opened for reading and/or writing as specified by the argument flags and the descriptor returned to the calling process.

Returns:

If successful, **open()** (p. 58) returns a non-negative integer, termed a device descriptor. It returns -1 on failure, and sets errno to indicate the error.

B.7. lcd.h File Reference

4x20 (4 lines, 20 chars per line) text LCD display **driver** (p. 24).

Defines

- #define **LCD_BASE_ADDRESS** 0xF0001050
- #define **LCD_CMD_ADDRESS** LCD_BASE_ADDRESS
- #define **LCD_DATA_ADDRESS** (LCD_BASE_ADDRESS + 8)

Functions

- void **lcd_init** (void)
- void **lcd_put_char** (unsigned char c)
- void **lcd_put_string** (char *s)
- void **lcd_put_chars** (char *data, int n)
- void **lcd_clear_display** (void)
- void **lcd_return_home** (void)
- void **lcd_set_cursor** (unsigned char addr)
- void **lcd_set_newline** (void)
- void **lcd_clear_addr_space** (unsigned char start_addr, unsigned char end_addr)
- void **lcd_send_byte_as_hex_string** (char byte)
- void **lcd_send_short_as_hex_string** (short sh)
- void **lcd_send_int_as_hex_string** (int i)
- void **hello_lcd** (void)
- void **uart2lcd_test** (void)

B.7.1. Detailed Description

Definition in file **lcd.h**.

B.7.2. Define Documentation

B.7.2.1. #define LCD_BASE_ADDRESS 0xF0001050

Definition at line 9 of file lcd.h.

B.7.2.2. #define LCD_CMD_ADDRESS LCD_BASE_ADDRESS

Definition at line 12 of file lcd.h.

B.7.2.3. #define LCD_DATA_ADDRESS (LCD_BASE_ADDRESS + 8)

Definition at line 13 of file lcd.h.

B.7.3. Function Documentation

B.7.3.1. void lcd_init (void)

Initialises the 4x20 (4 lines, 20 chars per line) text LCD (8-bit interface mode, display mode, dots format, visibility and blinking of cursor etc.). In order to use the display functions, you have to call this function for the initialisation of the device.

B.7.3.2. void lcd_put_char (unsigned char *c*)

Function for writing of char to lcd.

Writes a char (parameter c) to the next cursor position on the display. If the cursor has reached the line end, it automatically jumps to the beginning of the next line. In case it reaches the end of the display, it jumps to the start of the first line.

B.7.3.3. void lcd_put_string (char * *s*)

Function for writing of string to lcd.

Writes a complete string from the next cursor position on the display. Uses the function lcd_put_char internally.

B.7.3.4. void lcd_put_chars (char * *data*, int *n*)

Writes a fixed number of chars (param.: n) from the next cursor position on the display. Uses the function lcd_put_char internally.

B.7.3.5. void lcd_clear_display (void)

Clears the display and returns the cursor to the home position (beginning of the first line).

B.7.3.6. void lcd_return_home (void)

Resets the cursor to the home position (beginning of the first line).

B.7.3.7. void lcd_set_cursor (unsigned char *addr*)

Sets the cursor to the position addr. Contents are not deleted.

B.7.3.8. void lcd_set_newline (void)

Sets the cursor to the beginning of the next line. All remaining chars of the current line are deleted. This function is equivalent to a call of lcd_put_-char('\n').

B.7.3.9. void lcd_clear_addr_space (unsigned char *start_addr*, unsigned char *end_addr*)

Clears the display space between start_addr and end_addr (writes spaces on these positions).

B.7.3.10. void lcd_send_byte_as_hex_string (char *byte*)

Converts a byte to hexadecimal string and sends it to lcd.

B.7.3.11. void lcd_send_short_as_hex_string (short *sh*)

Converts a short to hexadecimal string and sends it to lcd.

B.7.3.12. void lcd_send_int_as_hex_string (int *i*)

Converts a int value to hexadecimal string and sends it to lcd.

B.7.3.13. void hello_lcd (void)

B.7.3.14. void uart2lcd_test (void)

B.8. log.h File Reference

Logging macros for the virtual output of the MERASA simulator.

```
#include <config.h>
#include <vo.h>
#include <pthread.h>
```

Defines

- #define **LOGLEVEL_DEBUG** 5
- #define **LOGLEVEL_INFO** 4
- #define **LOGLEVEL_WARN** 3
- #define **LOGLEVEL_ERR** 2
- #define **LOGLEVEL_FATAL** 1
- #define **LOGLEVEL_NONE** 0
- #define **LOGDBL_D4C**(w, c1, c2, c3, c4)
- #define **LOGDBL_X4C**(w, c1, c2, c3, c4)
- #define **log_debug_s**(msg, args...)
- #define **log_debug**(msg)
- #define **log_info_s**(msg, args...)
- #define **log_info**(msg)
- #define **log_warn_s**(msg, args...)
- #define **log_warn**(msg)
- #define **log_err_s**(msg, args...)
- #define **log_err**(msg)
- #define **log_fatal_s**(msg, args...)
- #define **log_fatal**(msg)

Functions

- int **sprintf** (char *str, const char *format,...)

Variables

- **pthread_mutex_t log_mutex**

B.8.1. Detailed Description

This file provides logging facilities by means of different loglevels (DEBUG, INFO, WARN, ERR, FATAL, NONE); the global loglevel is usually defined in

the Makefile. Attention: As the log output of a thread is locked by one common mutex, in some cases there may be problems like priority inversion. For this reason, we suggest to use the fast one-cycle logging methods LOGDBL_D4C, LOGDBL_DS, LOGDBL_X4C and LOGDBL_XS (for details of usage see below).

Definition in file **log.h**.

B.8.2. Define Documentation

B.8.2.1. #define LOGLEVEL_DEBUG 5

Definition at line 57 of file log.h.

B.8.2.2. #define LOGLEVEL_INFO 4

Definition at line 58 of file log.h.

B.8.2.3. #define LOGLEVEL_WARN 3

Definition at line 59 of file log.h.

B.8.2.4. #define LOGLEVEL_ERR 2

Definition at line 60 of file log.h.

B.8.2.5. #define LOGLEVEL_FATAL 1

Definition at line 61 of file log.h.

B.8.2.6. #define LOGLEVEL_NONE 0

Definition at line 62 of file log.h.

B.8.2.7. #define LOGDBL_D4C(w, c1, c2, c3, c4)

Value:

```
{ \
        uint64_t v = (w) | \
        ((uint64_t)(c1)<<32) | \
        ((uint64_t)(c2)<<40) | \
        ((uint64_t)(c3)<<48) | \
        ((uint64_t)(c4)<<56); \
        P64(VO_DBL_D4C) = v; \
}
```

Fast log of one decimal and four chars.

This macro enables a very fast logging output of one decimal and four chars (usually describing the decimal). As the output is finished within one cycle, you don't need the logging mutex and no priority inversion can occur.

Definition at line 97 of file log.h.

B.8.2.8. #define LOGDBL_X4C(w, c1, c2, c3, c4)

Value:

```
{ \
        uint64_t v = (w) | \
        ((uint64_t)(c1)<<32) | \
        ((uint64_t)(c2)<<40) | \
        ((uint64_t)(c3)<<48) | \
        ((uint64_t)(c4)<<56); \
        P64(VO_DBL_X4C) = v; \
}
```

Fast log of one decimal and a string.

This macro enables a very fast logging output of one decimal and a string. This string should contain 4 characters - if it's longer, it will be cut. As the output is finished within one cycle, you don't need the logging mutex and no priority inversion can occur.

Fast log of one hexadecimal and four chars This macro enables a very fast logging output of one hexadecimal and four chars (usually describing the hexadecimal). As the output is finished within one cycle, you don't need the logging mutex and no priority inversion can occur.

Definition at line 119 of file log.h.

B.8.2.9. #define log_debug_s(msg, args...)

Value:

```
{ \
        LOCK_LOG(); \
        char buffer[LOGBUFFLEN]; \
        sprintf(buffer, msg, args); \
        printf("[5,%u] %s:%d %s\n", get_thread(),__FILE__,  __LINE__, buffer); \
        UNLOCK_LOG(); \
}
```

Fast log of one hexadecimal and a string.

This macro enables a very fast logging output of one hexadecimal and a string. This string should contain 4 characters - if it's longer, it will be cut. As the output is finished within one cycle, you don't need the logging mutex and no priority inversion can occur.

Logging function for debug output with several arguments This macro enables a logging output for debugging. It is used similar to the common printf() with an undefined number of arguments.

Definition at line 149 of file log.h.

B.8.2.10. #define log_debug(msg)

Value:

```
{ \
        LOCK_LOG(); \
        printf("[5,%u] %s:%d %s\n", get_thread(),__FILE__,  __LINE__, msg); \
        UNLOCK_LOG(); \
}
```

Logging function for a string debug output.

This macro enables a logging output for debugging. It is used similar to the common printf(), but only for one argument.

Definition at line 166 of file log.h.

B.8.2.11. #define log_info_s(msg, args...)

Value:

```
{ \
        LOCK_LOG(); \
        char buffer[LOGBUFFLEN]; \
        sprintf(buffer, msg, args); \
        printf("[4,%u] %s:%d %s\n", get_thread(),__FILE__,  __LINE__, buffer); \
        UNLOCK_LOG(); \
}
```

Logging function for info output with several arguments.

This macro enables a logging output for information. It is used similar to the common printf() with an undefined number of arguments.

Definition at line 182 of file log.h.

B.8.2.12. #define log_info(msg)

Value:

```
{ \
        LOCK_LOG(); \
        printf("[4,%u] %s:%d %s\n", get_thread(),__FILE__,  __LINE__, msg); \
        UNLOCK_LOG(); \
}
```

Logging function for a string info output.

This macro enables a logging output for information. It is used similar to the common printf(), but only for one argument.

Definition at line 198 of file log.h.

B.8.2.13. #define log_warn_s(msg, args...)

Value:

```
{ \
        LOCK_LOG(); \
        char buffer[LOGBUFFLEN]; \
        sprintf(buffer, msg, args); \
        printf("[3,%u] %s:%d %s\n", get_thread(),__FILE__,  __LINE__, buffer); \
        UNLOCK_LOG(); \
}
```

Logging function for warning output with several arguments.

This macro enables a logging output for warning. It is used similar to the common printf() with an undefined number of arguments.

Definition at line 214 of file log.h.

B.8.2.14. #define log_warn(msg)

Value:

```
{ \
        LOCK_LOG(); \
        printf("[3,%u] %s:%d %s\n", get_thread(),__FILE__,  __LINE__, msg); \
        UNLOCK_LOG(); \
}
```

Logging function for a string warning output.

This macro enables a logging output for warning. It is used similar to the common printf(), but only for one argument.

Definition at line 230 of file log.h.

B.8.2.15. #define log_err_s(msg, args...)

Value:

```
{ \
        LOCK_LOG(); \
        char buffer[LOGBUFFLEN]; \
        sprintf(buffer, msg, args); \
        printf("[2,%u] %s:%d %s\n", get_thread(),__FILE__,  __LINE__, buffer); \
        UNLOCK_LOG(); \
}
```

Logging function for error output with several arguments.

This macro enables a logging output for error. It is used similar to the common printf() with an undefined number of arguments.

Definition at line 246 of file log.h.

B.8.2.16. #define log_err(msg)

Value:

```
{ \
        LOCK_LOG(); \
        printf("[2,%u] %s:%d %s\n", get_thread(),__FILE__,  __LINE__, msg); \
        UNLOCK_LOG(); \
}
```

Logging function for a string error output.

This macro enables a logging output for error. It is used similar to the common printf(), but only for one argument.

Definition at line 262 of file log.h.

B.8.2.17. #define log_fatal_s(msg, args...)

Value:

```
{ \
        LOCK_LOG(); \
        char buffer[LOGBUFFLEN]; \
        sprintf(buffer, msg, args); \
        printf("[1,%u] %s:%d %s\n", get_thread(),__FILE__,  __LINE__, buffer); \
        UNLOCK_LOG(); \
}
```

Logging function for fatal error output with several arguments.

This macro enables a logging output for fatal error. It is used similar to the common printf() with an undefined number of arguments.

Definition at line 278 of file log.h.

B.8.2.18. #define log_fatal(msg)

Value:

```
{ \
        LOCK_LOG(); \
        printf("[1,%u] %s:%d %s\n", get_thread(),__FILE__,  __LINE__, msg); \
        UNLOCK_LOG(); \
}
```

Logging function for a string fatal error output.

This macro enables a logging output for fatal error. It is used similar to the common printf(), but only for one argument.

Definition at line 294 of file log.h.

B.8.3. Function Documentation

B.8.3.1. int sprintf (char * *str*, const char * *format*, ...)

Better use the logging macros.

B.8.4. Variable Documentation

B.8.4.1. pthread_mutex_t log_mutex

The mutex for the virtual output of the MERASA simulator. Necessary, if you don't want a mixture of chars from different threads.

B.9. mcp2515.h File Reference

```
#include "spi.h"
#include "mcp2515_defs.h"
```

Data Structures

- struct **CANMessage_t**

Defines

- #define **MCP2515_INT_INPUT_PIO_BASE_ADDRESS** 0xF0001020
- #define **MCP2515_INT_INPUT_PIN_INDEX** 0x4
- #define **SELECT_MCP2515_CHIP** SPI_SELECT_SLAVE
- #define **DESELECT_MCP2515_CHIP** SPI_DESELECT_SLAVE
- #define **IS_MCP2515_INT_SET** ((*((char *) MCP2515_INT_INPUT_PIO_BASE_ADDRESS)) & (1<<MCP2515_INT_INPUT_PIN_INDEX))
- #define **MCP2515_CHECK_MESSAGE** (! IS_MCP2515_INT_SET)

Enumerations

- enum **can_bitrate_t** {

  CAN_BITRATE_10_KBPS, **CAN_BITRATE_20_KBPS**, **CAN_BITRATE_50_KBPS**, **CAN_BITRATE_100_KBPS**,

  CAN_BITRATE_125_KBPS, **CAN_BITRATE_250_KBPS**, **CAN_BITRATE_500_KBPS**, **CAN_BITRATE_1_MBPS** }
- enum **can_operation_mode_t** {

  CAN_CONFIGURATION_MODE, **CAN_NORMAL_MODE**, **CAN_SLEEP_MODE**, **CAN_LISTEN_ONLY_MODE**,

  CAN_LOOP_BACK_MODE }
- enum **can_receive_buffer_operation_mode_t** { **CAN_REC_BUFF_RECEIVE_STD_AND_EXT_ID_FILTER_MSG** = 0, **CAN_REC_BUFF_RECEIVE_ONLY_STD_ID_FILTER_MSG** = 1, **CAN_REC_BUFF_RECEIVE_ONLY_EXT_ID_FILTER_MSG** = 2, **CAN_REC_BUFF_RECEIVE_ANY_MSG** = 3 }
- enum **mcp2515_receive_buffer_t** { **MCP2515_REC_BUFF_0**, **MCP2515_REC_BUFF_1** }

- enum **mcp2515_filter_num_t** {
  CMCP2515_FILTER_0, **CMCP2515_FILTER_1**, **CMCP2515_FILTER_2**, **CMCP2515_FILTER_3**,
  CMCP2515_FILTER_4, **CMCP2515_FILTER_5** }
- enum **mcp2515_rollover_t** { **CMCP2515_ROLLOVER_DISABLE** = 0, **CMCP2515_ROLLOVER_ENABLE** = 1 }

Functions

- unsigned char **mcp2515_init** (**can_bitrate_t** bitrate)
- unsigned char **mcp2515_send_message** (**CANMessage_t** *message)
- char **mcp2515_get_message** (**CANMessage_t** *message)
- void **mcp2515_set_operation_mode** (**can_operation_mode_t** mode)
- void **mcp2515_set_mask** (**mcp2515_receive_buffer_t** buffer_num, unsigned short mask)
- void **mcp2515_set_filter** (**mcp2515_filter_num_t** filter_num, unsigned short filter)
- void **mcp2515_set_receive_buffer_operation_mode** (**mcp2515_receive_buffer_t** buffer_num, **can_receive_buffer_operation_mode_t** mode)
- void **mcp2515_set_rollover** (**mcp2515_rollover_t** rollover)
- void **print_can_message** (**CANMessage_t** *message)

B.9.1. Define Documentation

B.9.1.1. #define MCP2515_INT_INPUT_PIO_BASE_ADDRESS 0xF0001020

Definition at line 17 of file mcp2515.h.

B.9.1.2. #define MCP2515_INT_INPUT_PIN_INDEX 0x4

Definition at line 18 of file mcp2515.h.

B.9.1.3. #define SELECT_MCP2515_CHIP SPI_SELECT_SLAVE

Selects the MCP2515 Chip as SPI slave. In order to use this macro correctly, the macro **SPI_SET_SLAVE_SELECT_MASK(0)** (p. 103) should be called once during the initialisation phase of the MCP2515 controller.

Definition at line 26 of file mcp2515.h.

B.9.1.4. #define DESELECT_MCP2515_CHIP SPI_DESELECT_SLAVE

Deselects the MCP2515 Chip as SPI slave. In order to use this macro correctly, the macro **SPI_SET_SLAVE_SELECT_MASK(0)** (p. 103) should be called once during the initialisation phase of the MCP2515 controller.

Definition at line 32 of file mcp2515.h.

B.9.1.5. #define IS_MCP2515_INT_SET ((*((char *) MCP2515_INT_INPUT_PIO_BASE_ADDRESS)) & (1<<MCP2515_INT_INPUT_PIN_INDEX))

Tests whether the INT_n pin is set. The return value is not equal to 0, if this pin is set (high). Otherwise it returns 0.

Definition at line 37 of file mcp2515.h.

B.9.1.6. #define MCP2515_CHECK_MESSAGE (! IS_MCP2515_INT_SET)

Checks if there are any new messages waiting in the receive buffers, i.e. checks if the INT_n Pin is not set (is equal to 0).

Remark: In the current implementation there is only the receive interrupt enabled, therefore the INT_n pin is driven low by the MCP2515, only when an receive interrupt occurs.

Definition at line 45 of file mcp2515.h.

B.9.2. Enumeration Type Documentation

B.9.2.1. enum can_bitrate_t

Enumerator:

CAN_BITRATE_10_KBPS
CAN_BITRATE_20_KBPS
CAN_BITRATE_50_KBPS
CAN_BITRATE_100_KBPS
CAN_BITRATE_125_KBPS
CAN_BITRATE_250_KBPS
CAN_BITRATE_500_KBPS
CAN_BITRATE_1_MBPS

Definition at line 55 of file mcp2515.h.

B.9.2.2. enum can_operation_mode_t

Enumerator:

CAN_CONFIGURATION_MODE
CAN_NORMAL_MODE
CAN_SLEEP_MODE
CAN_LISTEN_ONLY_MODE
CAN_LOOP_BACK_MODE

Definition at line 66 of file mcp2515.h.

B.9.2.3. enum can_receive_buffer_operation_mode_t

Enumerator:

CAN_REC_BUFF_RECEIVE_STD_AND_EXT_ID_FILTER_MSG

CAN_REC_BUFF_RECEIVE_ONLY_STD_ID_FILTER_MSG
CAN_REC_BUFF_RECEIVE_ONLY_EXT_ID_FILTER_MSG
CAN_REC_BUFF_RECEIVE_ANY_MSG

Definition at line 74 of file mcp2515.h.

B.9.2.4. enum mcp2515_receive_buffer_t

Enumerator:

MCP2515_REC_BUFF_0
MCP2515_REC_BUFF_1

Definition at line 81 of file mcp2515.h.

B.9.2.5. enum mcp2515_filter_num_t

Enumerator:

CMCP2515_FILTER_0
CMCP2515_FILTER_1
CMCP2515_FILTER_2
CMCP2515_FILTER_3
CMCP2515_FILTER_4
CMCP2515_FILTER_5

Definition at line 86 of file mcp2515.h.

B.9.2.6. enum mcp2515_rollover_t

Enumerator:

CMCP2515_ROLLOVER_DISABLE

CMCP2515_ROLLOVER_ENABLE

Definition at line 95 of file mcp2515.h.

B.9.3. Function Documentation

B.9.3.1. unsigned char mcp2515_init (can_bitrate_t *bitrate*)

Initialises the MCP2515 Chip (sets bit rate, enables Receive Interrupt, sets the mode: "receive any message without filtering"). In order to use the CAN functions, you have to call this function for the initialisation of the device. The return value is equal to 0, if there was an error during the initialisation (bit rate unsupported or mcp2515 chip not accessible). The return value is equal to 1, if the initialisation was successful. Supported parameters are: BITRATE_10_KBPS; BITRATE_20_KBPS; BITRATE_50_KBPS; BITRATE_100_KBPS; BITRATE_125_KBPS; BITRATE_250_KBPS; BITRATE_500_KBPS; BITRATE_1_MBPS

B.9.3.2. unsigned char mcp2515_send_message (CANMessage_t * *message*)

Sends the CAN-message (parameter message) via CAN interface. The return value is equal to: 0, if all transmit buffer was full and therefore the CAN module could not send the message. 1, if the message to send was written to transmit buffer 0. 2, if the message to send was written to transmit buffer 1. 4, if the message to send was written to transmit buffer 2.

B.9.3.3. char mcp2515_get_message (CANMessage_t * *message*)

Checks according to the status of the CAN controller, if a new message is received in one of the receive buffers of the CAN controller. If yes, the message is copied to parameter message and the corresponding flag is deleted in the flag register of the CAN controller (by this, the message is no more valid, as it was already read). If both receive buffers got new messages, only the message of buffer 0 is read and only this flag is deleted. The second message in buffer 1 is not handled before the next call of this function. If there is no message in both receive buffers, the return value is equal to -1. A positive return value shows the filter matches (the number of the acceptance filter) on received messages.

All possible return values are:

- 1 = no message available 0 = filter 0 has accepted the message which is now in receive buffer 0 1 = filter 1 has accepted the message which is now in receive buffer 0 2 = filter 2 has accepted the message which is

now in receive buffer 1 3 = filter 3 has accepted the message which is now in receive buffer 1 4 = filter 4 has accepted the message which is now in receive buffer 1 5 = filter 5 has accepted the message which is now in receive buffer 1 6 = filter 0 has accepted the message; rollover to receive buffer 1 7 = filter 1 has accepted the message; rollover to receive buffer 1

For performance reasons it is recommended to use (before calling of this function) the macro MCP2515_CHECK_MESSAGE in order to check if a new message has been received.

B.9.3.4. void mcp2515_set_operation_mode (can_operation_mode_t *mode*)

Sets the operation mode of the mcp2515 controller specified by parameter mode. Five valid values for the parameter mode are: CAN_CONFIGURATION_MODE, CAN_NORMAL_MODE, CAN_SLEEP_MODE, CAN_LISTEN_ONLY_MODE, CAN_LOOP_BACK_MODE

B.9.3.5. void mcp2515_set_mask (mcp2515_receive_buffer_t *buffer_num*, unsigned short *mask*)

Sets the mask value (parameter mask) for the receive buffer specified by buffer_num. Two valid values for the parameter buffer_num are: MCP2515_REC_BUFF_0 (for buffer RXB0), MCP2515_REC_BUFF_1 (for buffer RXB1)

Restriction: sets masks only for Standard Identifiers.

B.9.3.6. void mcp2515_set_filter (mcp2515_filter_num_t *filter_num*, unsigned short *filter*)

Sets the filter value (par.: filter) for the acceptance filter specified by filter_num.

The valid values for the parameter filter_num are: CMCP2515_FILTER_0, (for receive buffer 0) CMCP2515_FILTER_1, (for receive buffer 0) CMCP2515_FILTER_2, (for receive buffer 1) CMCP2515_FILTER_3, (for receive buffer 1) CMCP2515_FILTER_4, (for receive buffer 1) CMCP2515_FILTER_5, (for receive buffer 1)

Restriction: sets filter only for Standard Identifiers.

B.9.3.7. void mcp2515_set_receive_buffer_operation_mode (mcp2515_receive_buffer_t *buffer_num*, can_receive_buffer_operation_mode_t *mode*)

Sets the operation mode for the receive buffer specified by buffer_num.

The valid values for the parameter buffer_num are: MCP2515_REC_BUFF_0 (for receive buffer 0), MCP2515_REC_BUFF_1 (for receive buffer 1)

Two valid values for parameter mode are: CAN_REC_BUFF_RECEIVE_-ONLY_STD_ID_FILTER_MSG (Receive only valid messages with standard identifiers that meet filter criteria) CAN_REC_BUFF_RECEIVE_ANY_-MSG (Turn mask and filters off; receive any message)

B.9.3.8. void mcp2515_set_rollover (mcp2515_rollover_t *rollover*)

Turns on/of the rollover mode. If rollover mode is on, then the new valid message of receive buffer 0 will be moved into receive buffer 1 when the receive buffer 0 already contains another valid message.

The values for the parameter rollover are: CMCP2515_ROLLOVER_-DISABLE CMCP2515_ROLLOVER_ENABLE

B.9.3.9. void print_can_message (CANMessage_t * *message*)

Prints the content of the CAN message over UART.

B.10. memory-desc.h File Reference

Description of memory.

`#include <sys/types.h>`

`#include <pthread.h>`

Defines

- #define **MEMORY_CFG**(b, l, ac, c, f)

Variables

- **memory_t node_mem_config** []
- const size_t **mem_config_len**

B.10.1. Detailed Description

This file provides functionalities to define and characterize a memory region.

Definition in file **memory-desc.h**.

B.10.2. Define Documentation

B.10.2.1. #define MEMORY_CFG(b, l, ac, c, f)

Value:

```
{       (b), \
        (b) + 1, \
        l, \
        ac, \
        c, \
        f, \
        0, \
        0 \
}
```

Use this macro to define a memory region. See an example of usage in the userapp.c.

Parameters:

b begin

l lenght

ac access_cycles

c cost

f flags

Definition at line 83 of file memory-desc.h.

B.10.3. Variable Documentation

B.10.3.1. memory_t node_mem_config[]

You need to define this constant in the OS for one specific node configuration. Use the **MEMORY_CFG** (p. 76) macro below for filling this array!

B.10.3.2. const size_t mem_config_len

For correct access to **node_mem_config** (p. 77), you have to set this constant as sizeof(node_mem_config)/sizeof(memory_t).

B.11. memory.h File Reference

Basic functionalities of memory management.

`#include <pthread.h>`

`#include <sys/types.h>`

`#include <memory-desc.h>`

Functions

- void **__init_malloc_shared** (void)
- void * **malloc_shared** (size_t size)
- void * **calloc_shared** (size_t number, size_t size)
- void * **realloc_shared** (void *ptr, size_t size)
- void **free_shared** (void *ptr)
- void * **malloc** (size_t size)
- void * **calloc** (size_t number, size_t size)
- void * **realloc** (void *ptr, size_t size)
- void **free** (void *ptr)
- void * **tcmemcpy** (void *dest, const void *src, size_t n)
- **bool_t has_write_permission** (void *mem)

B.11.1. Detailed Description

This file provides functionalities to allocate and free memory, but also to copy memory regions or check for write permissions.

Definition in file **memory.h**.

B.11.2. Function Documentation

B.11.2.1. void __init_malloc_shared (void)

Initialises memory for the shared allocation (size is defined by _SHARED_-MEM_LENGTH in userapp.c).

B.11.2.2. void* malloc_shared (size_t *size*)

The **malloc_shared()** (p. 78) function allocates size bytes of shared memory (from the shared heap for all threads) and returns a pointer to the allocated memory. The allocation is based on the TLSF allocator, i.e. real-time capable.

Parameters:

size amount of memory to allocate

B.11.2.3. void* calloc_shared (size_t *number*, size_t *size*)

The **calloc_shared()** (p. 79) function allocates space for number objects, each size bytes in length. The result is identical to calling **calloc_shared()** (p. 79) with an argument of "number * size", with the exception that the allocated memory is explicitly initialized to zero bytes.

B.11.2.4. void* realloc_shared (void * *ptr*, size_t *size*)

The **realloc_shared()** (p. 79) function changes the size of the previously allocated shared memory referenced by ptr to size bytes. The contents of the memory are unchanged up to the lesser of the new and old sizes. If the new size is larger, the value of the newly allocated portion of the memory is undefined. Upon success, the memory referenced by ptr is freed and a pointer to the newly allocated memory is returned.

B.11.2.5. void free_shared (void * *ptr*)

Free a previously allocated shared block of the current thread.

Parameters:

ptr the memory block

B.11.2.6. void* malloc (size_t *size*)

The **malloc()** (p. 79) function allocates size bytes of memory and returns a pointer to the allocated memory.

Parameters:

size amount of memory to allocate

B.11.2.7. void* calloc (size_t *number*, size_t *size*)

The **calloc()** (p. 79) function allocates space for number objects, each size bytes in length. The result is identical to calling **malloc()** (p. 79) with an argument of "number * size", with the exception that the allocated memory is explicitly initialized to zero bytes.

B.11.2.8. void* realloc (void * *ptr*, size_t *size*)

The **realloc()** (p. 79) function changes the size of the previously allocated memory referenced by ptr to size bytes. The contents of the memory are unchanged up to the lesser of the new and old sizes. If the new size is larger, the value of the newly allocated portion of the memory is undefined. Upon success, the memory referenced by ptr is freed and a pointer to the newly allocated memory is returned.

B.11.2.9. void free (void * *ptr*)

Free a previously allocated block of the current thread.

Parameters:

ptr the memory block

B.11.2.10. void* tcmemcpy (void * *dest*, const void * *src*, size_t *n*)

Effective copying of (non-overlapping!) memory.

B.11.2.11. bool_t has_write_permission (void * *mem*)

Check if the thread is allowed to write to the specified address.

B.12. merasa-ssw.h File Reference

All MERASA includes.

```
#include <error.h>
#include <driver.h>
#include <drivermanager.h>
#include <fcntl.h>
#include <log.h>
#include <memory-desc.h>
#include <memory.h>
#include <pthread.h>
#include <stropts.h>
#include <sysmonitor.h>
#include <sys/types.h>
#include <unistd.h>
#include <vo.h>
```

B.12.1. Detailed Description

This file contains all headers of the MERASA include directory. This enables an easy inclusion into your user application.

Definition in file **merasa-ssw.h**.

B.13. pio.h File Reference

Driver functions for the four user-defined push-button switches (sw4 to sw7), eight LEDs (D1 to D8) and a dual seven-segment display of the FPGA board. All these components are connected to the MERASA processor over a parallel input / output port (PIO).

Defines

- #define **INPUT_BUTTON_PIO_BASE_ADDRESS** 0xF0001020
- #define **LED_PIO_BASE_ADDRESS** 0xF0001030
- #define **DEBUG_OUTPUT_PIO_BASE_ADDRESS** 0xF0001040
- #define **input_button_pio** (char *) INPUT_BUTTON_PIO_BASE_ADDRESS
- #define **led_pio** (char *) LED_PIO_BASE_ADDRESS
- #define **debug_output_pio** (int *) DEBUG_OUTPUT_PIO_BASE_ADDRESS
- #define **LED1** (0x01)
- #define **LED2** (0x02)
- #define **LED3** (0x04)
- #define **LED4** (0x08)
- #define **LED5** (0x10)
- #define **LED6** (0x20)
- #define **LED7** (0x40)
- #define **LED8** (0x80)
- #define **LED1_ON** ((*led_pio) = LED1)
- #define **LED2_ON** ((*led_pio) = LED2)
- #define **LED3_ON** ((*led_pio) = LED3)
- #define **LED4_ON** ((*led_pio) = LED4)
- #define **LED5_ON** ((*led_pio) = LED5)
- #define **LED6_ON** ((*led_pio) = LED6)
- #define **LED7_ON** ((*led_pio) = LED7)
- #define **LED8_ON** ((*led_pio) = LED8)
- #define **OFF_LEDs** ((*led_pio) = 0x00)
- #define **SW4** (0x01)
- #define **SW5** (0x02)
- #define **SW6** (0x04)
- #define **SW7** (0x08)

Functions

- void **send_to_debug_output** (int output_value)
- void **sevenSegment_put_char** (unsigned char byte)
- void **set_LEDs** (unsigned char leds)
- char **read_user_buttons** (void)
- int **is_button_sw4_pressed** (void)
- int **is_button_sw5_pressed** (void)
- int **is_button_sw6_pressed** (void)
- int **is_button_sw7_pressed** (void)

B.13.1. Detailed Description

Definition in file **pio.h**.

B.13.2. Define Documentation

B.13.2.1. #define INPUT_BUTTON_PIO_BASE_ADDRESS 0xF0001020

Definition at line 11 of file pio.h.

B.13.2.2. #define LED_PIO_BASE_ADDRESS 0xF0001030

Definition at line 12 of file pio.h.

B.13.2.3. #define DEBUG_OUTPUT_PIO_BASE_ADDRESS 0xF0001040

Definition at line 13 of file pio.h.

B.13.2.4. #define input_button_pio (char *) INPUT_BUTTON_PIO_BASE_ADDRESS

Definition at line 16 of file pio.h.

B.13.2.5. #define led_pio (char *) LED_PIO_BASE_ADDRESS

Definition at line 17 of file pio.h.

B.13.2.6. #define debug_output_pio (int *) DEBUG_OUTPUT_PIO_BASE_ADDRESS

Definition at line 18 of file pio.h.

B.13.2.7. #define LED1 (0x01)

Definition at line 21 of file pio.h.

B.13.2.8. #define LED2 (0x02)

Definition at line 22 of file pio.h.

B.13.2.9. #define LED3 (0x04)

Definition at line 23 of file pio.h.

B.13.2.10. #define LED4 (0x08)

Definition at line 24 of file pio.h.

B.13.2.11. #define LED5 (0x10)

Definition at line 25 of file pio.h.

B.13.2.12. #define LED6 (0x20)

Definition at line 26 of file pio.h.

B.13.2.13. #define LED7 (0x40)

Definition at line 27 of file pio.h.

B.13.2.14. #define LED8 (0x80)

Definition at line 28 of file pio.h.

B.13.2.15. #define LED1_ON ((*led_pio) = LED1)

Definition at line 30 of file pio.h.

B.13.2.16. #define LED2_ON ((*led_pio) = LED2)

Definition at line 31 of file pio.h.

B.13.2.17. #define LED3_ON ((*led_pio) = LED3)

Definition at line 32 of file pio.h.

B.13.2.18. #define LED4_ON ((*led_pio) = LED4)

Definition at line 33 of file pio.h.

B.13.2.19. #define LED5_ON ((*led_pio) = LED5)

Definition at line 34 of file pio.h.

B.13.2.20. #define LED6_ON ((*led_pio) = LED6)

Definition at line 35 of file pio.h.

B.13.2.21. #define LED7_ON ((*led_pio) = LED7)

Definition at line 36 of file pio.h.

B.13.2.22. #define LED8_ON ((*led_pio) = LED8)

Definition at line 37 of file pio.h.

B.13.2.23. #define OFF_LEDs ((*led_pio) = 0x00)

Definition at line 39 of file pio.h.

B.13.2.24. #define SW4 (0x01)

Definition at line 43 of file pio.h.

B.13.2.25. #define SW5 (0x02)

Definition at line 44 of file pio.h.

B.13.2.26. #define SW6 (0x04)

Definition at line 45 of file pio.h.

B.13.2.27. #define SW7 (0x08)

Definition at line 46 of file pio.h.

B.13.3. Function Documentation

B.13.3.1. void send_to_debug_output (int *output_value*)

Sends the parameter output_value to the debug output port of the MERASA processor.

Currently the debug output of the MERASA processor is connected to the seven segment display controller.

B.13.3.2. void sevenSegment_put_char (unsigned char *byte*)

Writes an unsigned char value (parameter byte) in hexadecimal format on the dual seven segment display of the FPGA board.

B.13.3.3. void set_LEDs (unsigned char *leds*)

Switch all LEDs on or off, together with a combination of LED1, LED2, LED3 ... LED8. Example: setLeds(LED1 | LED3 | LED4 | LED8);

B.13.3.4. char read_user_buttons (void)

Reads all four user-defined push-button switches (sw4 to sw7).

The lowest 4 bits of the return value contain the state of the buttons. The lowest bit (bit 0) represents button sw4, the bit 1 corresponds to sw5 and so on. If the button is pressed, the corresponding bit value is set to the logical '0' otherwise '1'. Accordingly if all four buttons are pressed, then the function result is 0. If no button is pressed, then the result of this fuction is 0x0F.

(Press sw4) => (result = 0x0E); (Press sw7) => (result = 0x07); (Press sw4) and sw5 => (result = 0x0C); and so on.

B.13.3.5. int is_button_sw4_pressed (void)

Detects if the push-button sw4 of the FPGA board is pressed.

The return value of this function is 1, if botton sw4 is pressed, otherwise 0.

B.13.3.6. int is_button_sw5_pressed (void)

Detects if the push-button sw5 of the FPGA board is pressed.

The return value of this function is 2, if botton sw5 is pressed, otherwise 0.

B.13.3.7. int is_button_sw6_pressed (void)

Detects if the push-button sw6 of the FPGA board is pressed.

The return value of this function is 4, if sw6 botton is pressed, otherwise 0.

B.13.3.8. int is_button_sw7_pressed (void)

Detects if the push-button sw7 of the FPGA board is pressed.

The return value of this function is 8, if botton sw7 is pressed, otherwise 0.

B.14. pthread.h File Reference

POSIX thread functions and more.

`#include <sys/types.h>`

`#include <config.h>`

Defines

- #define **NO_THREAD** ((**int32_t**)-1)
- #define **NO_OWNER** ((**thread_handler**)-1)
- #define **get_thread**() get_current_thread_handler()
- #define **THREAD_PRIV_NRT** 0x00000001
- #define **THREAD_PRIV_HRT** 0x00000004
- #define **THREAD_PRIV_THRMG** 0x00000008
- #define **THREAD_PRIV_DYNMEM** 0x00000010
- #define **THREAD_PRIV_MEXT** 0x00000020
- #define **THREAD_PRIV_TLSF** 0x00000040
- #define **THREAD_PRIV_MODS** 0x00000100
- #define **THREAD_PRIV_GMOD** 0x00000200
- #define **THREAD_PRIV_APPS** 0x00000400
- #define **THREAD_PRIV_DRVS** 0x00000800

Functions

- **threadptr get_thread4handler** (**thread_handler** th)
- **threadptr get_current_thread** (void)
- **thread_handler get_current_thread_handler** (void)
- **bool_t has_privilege** (**uint32_t** priv)
- void **yield** (void)
- static **uint32_t get_current_core** (void)
- static **uint32_t get_current_slot** (void)
- **uint32_t finish_thread_init** (void)
- int **pthread_create** (**pthread_t** *thread, const **pthread_attr_t** *attr, void *(*start_routine)(void *), void *arg)
- int **pthread_join** (**pthread_t** thread, void **value_ptr)
- **pthread_t pthread_self** (void)
- void **pthread_yield** (void)
- int **pthread_attr_getschedparam** (const **pthread_attr_t** *attr, **sched_param** *param)
- int **pthread_attr_getschedpolicy** (const **pthread_attr_t** *attr, int

*policy)

- int **pthread_attr_init** (**pthread_attr_t** *attr)
- int **pthread_attr_setschedparam** (**pthread_attr_t** *attr, **sched_param** *param)
- int **pthread_attr_setschedpolicy** (**pthread_attr_t** *attr, int policy)
- int **pthread_attr_getmemory** (**pthread_attr_t** *attr, **memory_t** **mem)
- int **pthread_attr_setmemory** (**pthread_attr_t** *attr, **memory_t** *mem)
- int **pthread_attr_getbasicheapsize** (**pthread_attr_t** *attr, int *heapsize)
- int **pthread_attr_setbasicheapsize** (**pthread_attr_t** *attr, int heapsize)
- int **pthread_attr_getflags** (**pthread_attr_t** *attr, int *flags)
- int **pthread_attr_setflags** (**pthread_attr_t** *attr, int flags)
- int **pthread_attr_getiq** (**pthread_attr_t** *attr, int *iq)
- int **pthread_attr_setiq** (**pthread_attr_t** *attr, int iq)
- int **pthread_mutex_destroy** (**pthread_mutex_t** *mutex)
- int **pthread_mutex_init** (**pthread_mutex_t** *mutex, const **pthread_mutexattr_t** *attr)
- int **pthread_mutex_lock** (**pthread_mutex_t** *mutex)
- int **pthread_mutex_trylock** (**pthread_mutex_t** *mutex)
- int **pthread_mutex_unlock** (**pthread_mutex_t** *mutex)
- int **pthread_cond_broadcast** (**pthread_cond_t** *cond)
- int **pthread_cond_destroy** (**pthread_cond_t** *cond)
- int **pthread_cond_init** (**pthread_cond_t** *cond, const **pthread_condattr_t** *attr)
- int **pthread_cond_signal** (**pthread_cond_t** *cond)
- int **pthread_cond_wait** (**pthread_cond_t** *, **pthread_mutex_t** *mutex)
- int **pthread_barrier_destroy** (**pthread_barrier_t** *barrier)
- int **pthread_barrier_init** (**pthread_barrier_t** *barrier, const **pthread_barrierattr_t** *attr, unsigned count)
- int **pthread_barrier_wait** (**pthread_barrier_t** *barrier)

B.14.1. Detailed Description

This file contains not only a subset of POSIX thread functions, but also some other definitions and tool functions for thread management.

Definition in file **pthread.h**.

B.14.2. Define Documentation

B.14.2.1. #define NO_THREAD ((int32_t)-1)

Return value if no thread is selected.

Definition at line 51 of file pthread.h.

B.14.2.2. #define NO_OWNER ((thread_handler)-1)

Return value if no owner is selected.

Definition at line 53 of file pthread.h.

B.14.2.3. #define get_thread() get_current_thread_handler()

Returns the actual thread handler.

Definition at line 55 of file pthread.h.

B.14.2.4. #define THREAD_PRIV_NRT 0x00000001

Non real-time scheduling.

Definition at line 59 of file pthread.h.

B.14.2.5. #define THREAD_PRIV_HRT 0x00000004

Hard real-time scheduling. The hard realtime thread should use TLSF (real-time capable memory management).

Definition at line 61 of file pthread.h.

B.14.2.6. #define THREAD_PRIV_THRMG 0x00000008

Thread has access to thread management (usually only boot thread).

Definition at line 63 of file pthread.h.

B.14.2.7. #define THREAD_PRIV_DYNMEM 0x00000010

Thread uses dynamic memory management.

Definition at line 67 of file pthread.h.

B.14.2.8. #define THREAD_PRIV_MEXT 0x00000020

Thread may extend its memory (i.e. the local malloc may do so) (needs DYNMEM!).

Definition at line 69 of file pthread.h.

B.14.2.9. #define THREAD_PRIV_TLSF 0x00000040

Thread uses TLSF for DSA, or if not set, uses DLAlloc (needs DYNMEN!). When using TLSF, online-memory extension is not allowed (the MEXT flag is ignored). So make sure to reserve enough memory at thread creation!

Definition at line 71 of file pthread.h.

B.14.2.10. #define THREAD_PRIV_MODS 0x00000100

Thread is allowed to load program modules (needs DYNMEM and MEXT too!). Use with care! The local namespace will influence the memory consumption of the thread.

Definition at line 75 of file pthread.h.

B.14.2.11. #define THREAD_PRIV_GMOD 0x00000200

Thread may load modules into global namespace.

Definition at line 77 of file pthread.h.

B.14.2.12. #define THREAD_PRIV_APPS 0x00000400

Thread may load applications.

Definition at line 79 of file pthread.h.

B.14.2.13. #define THREAD_PRIV_DRVS 0x00000800

Thread may manage drivers (load/unload).

Definition at line 81 of file pthread.h.

B.14.3. Function Documentation

B.14.3.1. threadptr get_thread4handler (thread_handler *th*)

Returns the thread pointer of the specified thread.

B.14.3.2. threadptr get_current_thread (void)

Returns the thread pointer of the current thread.

B.14.3.3. thread_handler get_current_thread_handler (void)

Returns the thread handler of the current thread.

B.14.3.4. bool_t has_privilege (uint32_t *priv*)

Returns 1, if the current thread has a specified privilege. Otherwise 0.

B.14.3.5. void yield (void)

Allows the scheduler to run another thread instead of the current one.

B.14.3.6. static uint32_t get_current_core (void) [inline, static]

Returns the core of the current thread.

Definition at line 97 of file pthread.h.

References MSS_MY_CORE.

B.14.3.7. static uint32_t get_current_slot (void) [inline, static]

Returns the slot of the current thread.

Definition at line 101 of file pthread.h.

References MSS_MY_SLOT.

B.14.3.8. uint32_t finish_thread_init (void)

Finishes the initialisation phase.

This function finishes the thread creation phase and sets the lists of hard- and non real-time threads to the scheduler. Attention: The scheduling will not start before calling this function!

B.14.3.9. int pthread_create (pthread_t * *thread*, const pthread_attr_t * *attr*, void *(*)(void *) *start_routine*, void * *arg*)

Creates a new thread of execution.

The **pthread_create()** (p. 91) function is used to create a new thread, with attributes specified by attr, within a process. If the attributes specified by attr are modified later, the thread's attributes are not affected. Upon successful completion **pthread_create()** (p. 91) will store the ID of the created thread in the location specified by thread. The thread is created executing start_routine with arg as its sole argument.

Returns:

If successful, the **pthread_create()** (p. 91) function will return zero. Otherwise an error number will be returned to indicate the error.

B.14.3.10. int pthread_join (pthread_t *thread*, void ** *value_ptr*)

Causes the calling thread to wait for the termination of the specified thread.

The **pthread_join()** (p. 91) function suspends execution of the calling thread until the target thread terminates unless the target thread has already terminated.

When a **pthread_join()** (p. 91) returns successfully, the target thread has been terminated. The results of multiple simultaneous calls to **pthread_join()** (p. 91) specifying the same tar- get thread are undefined. If the thread calling **pthread_join()** (p. 91) is can- celled, then the target thread is not detached.

Returns:

If successful, the **pthread_join()** (p. 91) function will return zero. Otherwise an error number will be returned to indicate the error.

B.14.3.11. pthread_t pthread_self (void)

Returns the thread ID of the calling thread.

Returns:

The **pthread_self()** (p. 92) function returns the thread ID of the calling thread.

B.14.3.12. void pthread_yield (void)

Allows the scheduler to run another thread instead of the current one.

B.14.3.13. int pthread_attr_getschedparam (const pthread_attr_t * *attr*, sched_param * *param*)

Get the scheduling parameter attribute from a thread attributes object.

B.14.3.14. int pthread_attr_getschedpolicy (const pthread_attr_t * *attr*, int * *policy*)

Get the scheduling policy attribute from a thread attributes object.

B.14.3.15. int pthread_attr_init (pthread_attr_t * *attr*)

Initialize a thread attributes object with default values.

Thread attributes are used to specify parameters to **pthread_create()** (p. 91). One attribute object can be used in multiple calls to **pthread_create()** (p. 91), with or without modifications between calls.

The **pthread_attr_init()** (p. 92) function initializes attr with all the default thread attributes.

Returns:

If successful, this function returns 0. Otherwise, an error number is returned to indicate the error.

B.14.3.16. int pthread_attr_setschedparam (pthread_attr_t * *attr*, sched_param * *param*)

Set the scheduling parameter attribute in a thread attributes object.

B.14.3.17. int pthread_attr_setschedpolicy (pthread_attr_t * *attr*, int *policy*)

Set the scheduling policy attribute in a thread attributes object.

This is necessary to select the right thread type. Possible values are

- `SCHED_HRT`: hard real-time thread
- `SCHED_NRT`: non real-time thread

After initialization, the policy is set to `SCHED_NONE` (a thread is not able to run with this value).

Returns:

If successful, this function returns 0. Otherwise, an error number is returned to indicate the error.

B.14.3.18. int pthread_attr_getmemory (pthread_attr_t * *attr*, memory_t ** *mem*)

Get the memory type used for the thread.

Returns:

If successful, this function returns 0. Otherwise, an error number is returned to indicate the error.

B.14.3.19. int pthread_attr_setmemory (pthread_attr_t * *attr*, memory_t * *mem*)

Set the memory to use for the thread.

If set to NULL, system standard memory will be used.

Returns:

If successful, this function returns 0. Otherwise, an error number is returned to indicate the error.

B.14.3.20. int pthread_attr_getbasicheapsize (pthread_attr_t * *attr*, int * *heapsize*)

Get the initial heap size used for the thread.

See **pthread_attr_setbasicheapsize()** (p. 94) for more details.

Returns:

If successful, this function returns 0. Otherwise, an error number is returned to indicate the error.

B.14.3.21. int pthread_attr_setbasicheapsize (pthread_attr_t * *attr*, int *heapsize*)

Set the initial heap size used for the thread.

Set the initial memory for the thread (only sensible, if the thread needs not to extend its memory). Calculation of the parameter thread_mem: For each variable, you need to allocate, add 4 bytes (1 word) management overhead and round each of these values up to a 8-byte-alignment. The minimum amount of memory that can be allocated is 16 bytes (4 word).

sz: amount of memory needed for a variable

=> real_sz = (sz+4 + 7) & ~8

Thus, all real_sz values added up result in the thread_mem parameter.

Returns:

If successful, this function returns 0. Otherwise, an error number is returned to indicate the error.

B.14.3.22. int pthread_attr_getflags (pthread_attr_t * *attr*, int * *flags*)

Get thread flags describing the thread's behaviour.

See **pthread_attr_setflags()** (p. 94) for an explanation of possible values.

Returns:

If successful, this function returns 0. Otherwise, an error number is returned to indicate the error.

B.14.3.23. int pthread_attr_setflags (pthread_attr_t * *attr*, int *flags*)

Set thread flags describing the thread's behaviour.

Possible values are:

- `THREAD_PRIV_THRMG`: thread has access to thread management (creating, killing, renicing)
- `THREAD_PRIV_DYNMEM`: thread uses dynamic memory management
- `THREAD_PRIV_MEXT`: thread my extend its memory (i.e. the local malloc may do so)
- `THREAD_PRIV_TLSF`: thread uses TLSF for DSA, or if not set, uses DLAlloc (needs DYNMEN!) When using TLSF, online-memory extension is not allowed (the MEXT flag is ignored). So make sure to reserve enough memory at thread creation!
- `THREAD_PRIV_MODS`: thread is allowed to load program modules (needs

DYNMEM and MEXT too!). Use with care! The local namespace will influence the memory consumption of the thread.

- `THREAD_PRIV_GMOD`: thread may load modules into global namespace
- `THREAD_PRIV_APPS`: thread may load applications
- `THREAD_PRIV_DRVS`: thread may manage drivers (load/unload)

Returns:

If successful, this function returns 0. Otherwise, an error number is returned to indicate the error.

B.14.3.24. int pthread_attr_getiq (pthread_attr_t * *attr*, int * *iq*)

Get the instruction quantum.

Returns:

If successful, this function returns 0. Otherwise, an error number is returned to indicate the error.

B.14.3.25. int pthread_attr_setiq (pthread_attr_t * *attr*, int *iq*)

Set the instruction quantum.

This value is given to the hardware-scheduler to decide how to schedule the different threads.

Returns:

If successful, this function returns 0. Otherwise, an error number is returned to indicate the error.

B.14.3.26. int pthread_mutex_destroy (pthread_mutex_t * *mutex*)

Destroy a mutex.

The **pthread_mutex_destroy()** (p. 95) function frees the resources allocated for mutex.

Returns:

If successful, **pthread_mutex_destroy()** (p. 95) will return zero, otherwise an error number will be returned to indicate the error.

B.14.3.27. int pthread_mutex_init (pthread_mutex_t * *mutex*, const pthread_mutexattr_t * *attr*)

Initialize a mutex with specified attributes.

The **pthread_mutex_init()** (p. 95) function creates a new mutex, with attributes specified with attr. If attr is NULL the default attributes are used.

Returns:

If successful, **pthread_mutex_init()** (p. 95) will return zero and put the new mutex id into mutex, otherwise an error number will be returned to indicate the error.

B.14.3.28. int pthread_mutex_lock (pthread_mutex_t * *mutex*)

Lock a mutex and block until it becomes available.

The **pthread_mutex_lock()** (p. 96) function locks mutex. If the mutex is already locked, the calling thread will block until the mutex becomes available.

Returns:

If successful, **pthread_mutex_lock()** (p. 96) will return zero, otherwise an error number will be returned to indicate the error.

B.14.3.29. int pthread_mutex_trylock (pthread_mutex_t * *mutex*)

Try to lock a mutex, but do not block if the mutex is locked by another thread, including the current thread.

The **pthread_mutex_trylock()** (p. 96) function locks mutex. If the mutex is already locked, **pthread_mutex_trylock()** (p. 96) will not block waiting for the mutex, but will return an error condition.

Returns:

If successful, **pthread_mutex_trylock()** (p. 96) will return zero, otherwise an error number will be returned to indicate the error.

B.14.3.30. int pthread_mutex_unlock (pthread_mutex_t * *mutex*)

Unlock a mutex.

If the current thread holds the lock on mutex, then the **pthread_mutex_unlock()** (p. 96) function unlocks mutex.

Returns:

If successful, **pthread_mutex_unlock()** (p. 96) will return zero, otherwise an error number will be returned to indicate the error.

B.14.3.31. int pthread_cond_broadcast (pthread_cond_t * *cond*)

Unblock all threads currently blocked on the specified condition variable.

The **pthread_cond_broadcast()** (p. 96) function unblocks all threads waiting for the condition variable cond.

Returns:

If successful, the **pthread_cond_broadcast()** (p. 96) function will return zero, otherwise an error number will be returned to indicate the error.

B.14.3.32. int pthread_cond_destroy (pthread_cond_t * *cond*)

Destroy a condition variable.

The **pthread_cond_destroy()** (p. 97) function frees the resources allocated by the condition variable cond.

Returns:

If successful, the **pthread_cond_destroy()** (p. 97) function will return zero, otherwise an error number will be returned to indicate the error.

B.14.3.33. int pthread_cond_init (pthread_cond_t * *cond*, const pthread_condattr_t * *attr*)

Initialize a condition variable with specified attributes.

The **pthread_cond_init()** (p. 97) function creates a new condition variable, with attributes specified with attr. If attr is NULL the default attributes are used.

Returns:

If successful, the **pthread_cond_init()** (p. 97) function will return zero and put the new condition variable id into cond, otherwise an error number will be returned to indicate the error.

B.14.3.34. int pthread_cond_signal (pthread_cond_t * *cond*)

Unblock at least one of the threads blocked on the specified condition variable.

The **pthread_cond_signal()** (p. 97) function unblocks one thread waiting for the condition variable cond.

Returns:

If successful, the **pthread_cond_signal()** (p. 97) function will return zero, otherwise an error number will be returned to indicate the error.

B.14.3.35. int pthread_cond_wait (pthread_cond_t *, pthread_mutex_t * *mutex*)

Wait for a condition and lock the specified mutex.

The **pthread_cond_wait()** (p. 97) function atomically blocks the current thread waiting on the condition variable specified by cond, and releases the mutex specified by mutex. The waiting thread unblocks only after another thread calls **pthread_cond_signal()** (p. 97), or **pthread_cond_broadcast()** (p. 96) with the same condition variable, and the current thread reacquires the lock on mutex.

Returns:

If successful, the **pthread_cond_wait()** (p. 97) function will return zero. Other- wise an error number will be returned to indicate the error.

B.14.3.36. int pthread_barrier_destroy (pthread_barrier_t * *barrier*)

B.14.3.37. int pthread_barrier_init (pthread_barrier_t * *barrier*, const pthread_barrierattr_t * *attr*, unsigned *count*)

B.14.3.38. int pthread_barrier_wait (pthread_barrier_t * *barrier*)

B.15. spi.h File Reference

Functions for sending/receiving from/to SPI (Serial Peripheral Interface).

Defines

- #define **SPI_BASE_ADDRESS** 0xF0001080
- #define **SPI_rxdata_offset** 0x0
- #define **SPI_txdata_offset** 0x4
- #define **SPI_status_offset** 0x8
- #define **SPI_control_offset** 0xC
- #define **SPI_slaveselect_offset** 0x14
- #define **SPI_RXDATA** *((volatile unsigned int *) (SPI_BASE_ADDRESS + SPI_rxdata_offset))
- #define **SPI_TXDATA** *((unsigned int *) (SPI_BASE_ADDRESS + SPI_txdata_offset))
- #define **SPI_STATUS** *((volatile unsigned int *) (SPI_BASE_ADDRESS + SPI_status_offset))
- #define **SPI_CONTROL** *((volatile unsigned int*) (SPI_BASE_ADDRESS + SPI_control_offset))
- #define **SPI_SLAVESELECT** *((unsigned int*) (SPI_BASE_ADDRESS + SPI_slaveselect_offset))
- #define **spi_ROE_bit_idx** 3
- #define **spi_TOE_bit_idx** 4
- #define **spi_TMT_bit_idx** 5
- #define **spi_TRDY_bit_idx** 6
- #define **spi_RRDY_bit_idx** 7
- #define **spi_E_bit_idx** 8
- #define **spi_SSO_bit_idx** 10
- #define **IS_SPI_DATA_RECEIVED** (SPI_STATUS & (1<<spi_RRDY_bit_idx))
- #define **IS_SPI_RECEIVE_OVERRUN_ERROR_OCCURED** (SPI_STATUS & (1<<spi_ROE_bit_idx))
- #define **IS_SPI_TRANSMITTER_READY** (SPI_STATUS & (1<<spi_TRDY_bit_idx))
- #define **IS_SPI_TRANSMITTER_OVERRUN_ERROR_OCCURED** (SPI_STATUS & (1<<spi_TOE_bit_idx))
- #define **SPI_CLEAR_ERROR_BITS** (SPI_STATUS = 0)
- #define **SPI_SET_SLAVE_SELECT_MASK**(slave_no) (SPI_SLAVESELECT = (1 << slave_no))
- #define **SPI_SELECT_SLAVE** (SPI_CONTROL |= (1<<spi_-

SSO_bit_idx))

- #define **SPI_DESELECT_SLAVE** (SPI_CONTROL &= ~(1<<spi_SSO_bit_idx))
- #define **SPI_PUT_CHAR**(char) while (! IS_SPI_TRANSMITTER_READY); \ SPI_TXDATA = char;

Functions

- void **spi_put_char** (char c)
- char **spi_get_char** (void)

B.15.1. Detailed Description

Definition in file **spi.h**.

B.15.2. Define Documentation

B.15.2.1. #define SPI_BASE_ADDRESS 0xF0001080

Definition at line 10 of file spi.h.

B.15.2.2. #define SPI_rxdata_offset 0x0

Definition at line 15 of file spi.h.

B.15.2.3. #define SPI_txdata_offset 0x4

Definition at line 16 of file spi.h.

B.15.2.4. #define SPI_status_offset 0x8

Definition at line 17 of file spi.h.

B.15.2.5. #define SPI_control_offset 0xC

Definition at line 18 of file spi.h.

B.15.2.6. #define SPI_slaveselect_offset 0x14

Definition at line 19 of file spi.h.

B.15.2.7. #define SPI_RXDATA *((volatile unsigned int *) (SPI_BASE_ADDRESS + SPI_rxdata_offset))

Definition at line 23 of file spi.h.

B.15.2.8. #define SPI_TXDATA *((unsigned int *) (SPI_BASE_ADDRESS + SPI_txdata_offset))

Definition at line 24 of file spi.h.

B.15.2.9. #define SPI_STATUS *((volatile unsigned int *) (SPI_BASE_ADDRESS + SPI_status_offset))

Definition at line 25 of file spi.h.

B.15.2.10. #define SPI_CONTROL *((volatile unsigned int*) (SPI_BASE_ADDRESS + SPI_control_offset))

Definition at line 26 of file spi.h.

B.15.2.11. #define SPI_SLAVESELECT *((unsigned int*) (SPI_BASE_ADDRESS + SPI_slaveselect_offset))

Definition at line 27 of file spi.h.

B.15.2.12. #define spi_ROE_bit_idx 3

Definition at line 30 of file spi.h.

B.15.2.13. #define spi_TOE_bit_idx 4

Definition at line 31 of file spi.h.

B.15.2.14. #define spi_TMT_bit_idx 5

Definition at line 32 of file spi.h.

B.15.2.15. #define spi_TRDY_bit_idx 6

Definition at line 33 of file spi.h.

B.15.2.16. #define spi_RRDY_bit_idx 7

Definition at line 34 of file spi.h.

B.15.2.17. #define spi_E_bit_idx 8

Definition at line 35 of file spi.h.

B.15.2.18. #define spi_SSO_bit_idx 10

Definition at line 39 of file spi.h.

B.15.2.19. #define IS_SPI_DATA_RECEIVED (SPI_STATUS & (1<<spi_RRDY_bit_idx))

Tests whether a new character is fully received via SPI. If it is received completely, the return value is not equal to 0. Otherwise the return value is equal to 0.

A master peripheral reads received data from the rxdata register. When the receive shift register receives a full n bits of data, the status registers RRDY bit is set to 1 and the data is transferred into the rxdata register.

REMARK: When software reads a value from the rxdata register, the status register's RRDY bit is set to 0 by hardware.

Definition at line 53 of file spi.h.

B.15.2.20. #define IS_SPI_RECEIVE_OVERRUN_ERROR_OCCURED (SPI_STATUS & (1<<spi_ROE_bit_idx))

Tests whether a receive-overrun error occured (return value 0 means "not occured").

If RRDY is 1 when data is transferred into the rxdata register (that is, the previous data was not retrieved), a receive-overrun error occurs and the status register's ROE bit is set to 1. In this case, the contents of rxdata are undefined.

Definition at line 63 of file spi.h.

B.15.2.21. #define IS_SPI_TRANSMITTER_READY (SPI_STATUS & (1<<spi_TRDY_bit_idx))

Tests whether the SPI transmitter is ready for sending a new character (return value 0 means "not ready").

I.e. tests whether the transmitter register (txdata) is ready for new data.

Characters should not be written to the txdata register until the transmitter is ready for new data, as indicated by the TRDY bit in the status register.

Definition at line 73 of file spi.h.

B.15.2.22. #define IS_SPI_TRANSMITTER_OVERRUN_ERROR_OCCURED (SPI_STATUS & (1<<spi_TOE_bit_idx))

Tests whether a transmit-overrun error occured (return value 0 means "not occured"). – This happens if TRDY is 0 and the MERASA processor writes new data to the txdata register for sending. In this case, the new data is ignored, and the content of the txdata register remains unchanged.

Definition at line 82 of file spi.h.

B.15.2.23. #define SPI_CLEAR_ERROR_BITS (SPI_STATUS = 0)

Clears the error bits (ROE, TOE and E bits) of the status register.

Definition at line 87 of file spi.h.

B.15.2.24. #define SPI_SET_SLAVE_SELECT_MASK(slave_no) (SPI_SLAVESELECT = (1 << slave_no))

Parameter slave_no specifies an SPI slave which should be selected by the macro SPI_SELECT_SLAVE. Slave numbers start with 0. As the MERASA SPI master currently communicates only with one single SPI slave (MCP2515 chip), this macro should be called only once with parameter slave_no = 0.

Definition at line 94 of file spi.h.

B.15.2.25. #define SPI_SELECT_SLAVE (SPI_CONTROL |= (1<<spi_SSO_bit_idx))

Forces the SPI core to drive its ss_n outputs (forces chipselect) , regardless of whether a serial shift operation is in progress or not. In order to specify which slave (ss_n) is selected, the corresponding bit should be set in slaveselect register in advance. For this purpose the macro SPI_SET_SLAVE_SELECT_MASK can be used.

Please note: As the MERASA SPI master currently communicates only with one single SPI slave (MCP2515 chip), this macro was implemented without parameter slave_number.

Definition at line 105 of file spi.h.

B.15.2.26. #define SPI_DESELECT_SLAVE (SPI_CONTROL &= ~(1<<spi_SSO_bit_idx))

Deselects the slave specified in slaveselect register.

Definition at line 110 of file spi.h.

B.15.2.27. #define SPI_PUT_CHAR(char) while (! IS_SPI_TRANSMITTER_READY); \ SPI_TXDATA = char;

Sends a char (one byte) over SPI; blocking until the transmitter is ready to send.

Definition at line 116 of file spi.h.

B.15.3. Function Documentation

B.15.3.1. void spi_put_char (char *c*) [inline]

Sends a char (one byte) over SPI; blocking until the transmitter is ready to send.

B.15.3.2. char spi_get_char (void) [inline]

Reads a new character (one byte) from rxdata register; blocking until a new character is fully received.

B.16. stropts.h File Reference

Control a device.

Functions

- int **ioctl** (int d, int request,...)

B.16.1. Detailed Description

Definition in file **stropts.h**.

B.16.2. Function Documentation

B.16.2.1. int ioctl (int *d*, int *request*, ...)

Control a device.

The **ioctl()** (p. 105) system call manipulates the underlying device parameters of special files. In particular, many operating characteristics of character special files (e.g. terminals) may be controlled with **ioctl()** (p. 105) requests. The argument d must be an open file descriptor.

Valid values for request and further parameters depend on the specific device, see the driver's header file.

Returns:

If an error has occurred, a value of -1 is returned and errno is set to indicate the error.

B.17. sysmonitor.h File Reference

Type definitions to describe the state of memory.

`#include <sys/types.h>`

Data Structures

- struct **memorystatistics**
- struct **thread_memorystatistics**

Typedefs

- typedef struct **memorystatistics** * **memstatptr**
- typedef struct **thread_memorystatistics** * **tmemstatptr**

B.17.1. Detailed Description

Definition in file **sysmonitor.h**.

B.17.2. Typedef Documentation

B.17.2.1. typedef struct memorystatistics* memstatptr

A shortcut

Definition at line 68 of file sysmonitor.h.

B.17.2.2. typedef struct thread_memorystatistics* tmemstatptr

A shortcut

Definition at line 84 of file sysmonitor.h.

B.18. timer.h File Reference

Functions for starting, reading and stopping of the Timer. And some auxiliary functions to convert the timer value into microseconds, milliseconds and seconds.

Defines

- #define **TIMER_BASE_ADDRESS** 0xF0001060
- #define **TIMER_READ_LO_ADDRESS** TIMER_BASE_ADDRESS
- #define **TIMER_READ_HI_ADDRESS** (TIMER_BASE_ADDRESS + 4)
- #define **TIMER_STOP_ADDRESS** (TIMER_BASE_ADDRESS + 8)
- #define **TIMER_START_ADDRESS** (TIMER_BASE_ADDRESS + 12)

Functions

- unsigned int **start_timer** (void)
- unsigned int **stop_timer** (void)
- unsigned int **read_timer32** (void)
- unsigned long long **read_timer48** (void)
- unsigned int **conv_clock_count_to_microsec** (unsigned int clock_count32)
- unsigned long long **conv_clock_count48_to_microsec** (unsigned long long clock_count48)
- unsigned int **conv_clock_count_to_millisec** (unsigned int clock_count32)
- unsigned int **conv_clock_count48_to_millisec** (unsigned long long clock_count48)
- unsigned int **conv_clock_count_to_sec** (unsigned int clock_count32)
- unsigned int **conv_clock_count48_to_sec** (unsigned long long clock_count48)

B.18.1. Detailed Description

Definition in file **timer.h**.

B.18.2. Define Documentation

B.18.2.1. #define TIMER_BASE_ADDRESS 0xF0001060

Definition at line 10 of file timer.h.

B.18.2.2. #define TIMER_READ_LO_ADDRESS TIMER_BASE_ADDRESS

Definition at line 13 of file timer.h.

B.18.2.3. #define TIMER_READ_HI_ADDRESS (TIMER_BASE_ADDRESS + 4)

Definition at line 14 of file timer.h.

B.18.2.4. #define TIMER_STOP_ADDRESS (TIMER_BASE_ADDRESS + 8)

Definition at line 15 of file timer.h.

B.18.2.5. #define TIMER_START_ADDRESS (TIMER_BASE_ADDRESS + 12)

Definition at line 16 of file timer.h.

B.18.3. Function Documentation

B.18.3.1. unsigned int start_timer (void)

Starts (or restarts) the timer. The return value is the clock count (32 Bits) between previous start and this restarting.

B.18.3.2. unsigned int stop_timer (void)

Stops the timer ie. stops the increasing of the clock counter. The return value is the clock count (32 Bits) between previous start and this stopping.

B.18.3.3. unsigned int read_timer32 (void)

The return value is the current timer value (32 Bits) ie. number of clock cycles after the start (or previous restart) of the timer. The counting is continued.

B.18.3.4. unsigned long long read_timer48 (void)

The return value is the current timer value (48 Bits) ie. number of clock cycles after the start (or previous restart) of the timer. The counting is continued.

B.18.3.5. unsigned int conv_clock_count_to_microsec (unsigned int *clock_count32*)

Converts the clock count (32 bits) into microseconds using the constant CLOCK_FREQUENCY_MHZ defined in clock.h

B.18.3.6. unsigned long long conv_clock_count48_to_microsec (unsigned long long *clock_count48*)

Converts the clock count (48 bits) into microseconds using the constant CLOCK_FREQUENCY_MHZ defined in clock.h

B.18.3.7. unsigned int conv_clock_count_to_millisec (unsigned int *clock_count32*)

Converts the clock count (32 bits) into milliseconds using the constant CLOCK_FREQUENCY_MHZ defined in clock.h

B.18.3.8. unsigned int conv_clock_count48_to_millisec (unsigned long long *clock_count48*)

Converts the clock count (48 bits) into milliseconds using the constant CLOCK_FREQUENCY_MHZ defined in clock.h

B.18.3.9. unsigned int conv_clock_count_to_sec (unsigned int *clock_count32*)

Converts the clock count (32 bits) into seconds using the constant CLOCK_FREQUENCY_MHZ defined in clock.h

B.18.3.10. unsigned int conv_clock_count48_to_sec (unsigned long long *clock_count48*)

Converts the clock count (48 bits) into seconds using the constant CLOCK_FREQUENCY_MHZ defined in clock.h

B.19. types.h File Reference

Definitions of basic data types.

`#include <stddef.h>`

Data Structures

- struct **sched_param**
- struct **pthread_mutex**
- struct **pthread_mutexattr_t**
- struct **pthread_cond**
- struct **pthread_condattr_t**
- struct **pthread_barrier_t**
- struct **pthread_barrierattr_t**
- struct **mem_cfg_data**
- struct **pthread_attr_t**

Defines

- #define **false** FALSE
- #define **true** TRUE

Typedefs

- typedef unsigned char **uint8_t**
- typedef signed char **int8_t**
- typedef unsigned short int **uint16_t**
- typedef signed short int **int16_t**
- typedef unsigned int **uint32_t**
- typedef signed int **int32_t**
- typedef unsigned long long **uint64_t**
- typedef signed long long **int64_t**
- typedef char * **address**
- typedef **int32_t error_t**
- typedef **uint32_t version_t**
- typedef signed int **ssize_t**
- typedef **int32_t thread_handler**
- typedef struct thread * **threadptr**
- typedef struct thread_control_block_t **tcb_t**
- typedef **uint32_t sched_t**

- typedef **int32_t cc_spinlock_t**
- typedef struct **pthread_mutex pthread_mutex_t**
- typedef struct **pthread_cond pthread_cond_t**
- typedef struct **mem_cfg_data memory_t**
- typedef **thread_handler pthread_t**

Enumerations

- enum **bool_t** { **FALSE** = 0, **TRUE** = 1 }
- enum **sched_policy** { **SCHED_NONE** = 0, **SCHED_HRT**, **SCHED_NRT** }

B.19.1. Detailed Description

Definition in file **types.h**.

B.19.2. Define Documentation

B.19.2.1. #define false FALSE

Definition at line 77 of file types.h.

B.19.2.2. #define true TRUE

Definition at line 78 of file types.h.

B.19.3. Typedef Documentation

B.19.3.1. typedef unsigned char uint8_t

Unsigned 8-bit integer

Definition at line 48 of file types.h.

B.19.3.2. typedef signed char int8_t

Signed 8-bit integer

Definition at line 50 of file types.h.

B.19.3.3. typedef unsigned short int uint16_t

Unsigned 16-bit integer

Definition at line 52 of file types.h.

B.19.3.4. typedef signed short int int16_t

Signed 16-bit integer

Definition at line 54 of file types.h.

B.19.3.5. typedef unsigned int uint32_t

Unsigned 32-bit integer

Definition at line 56 of file types.h.

B.19.3.6. typedef signed int int32_t

Signed 32-bit integer

Definition at line 58 of file types.h.

B.19.3.7. typedef unsigned long long uint64_t

Unsigned 64-bit integer

Definition at line 60 of file types.h.

B.19.3.8. typedef signed long long int64_t

Signed 64-bit integer

Definition at line 62 of file types.h.

B.19.3.9. typedef char* address

Data type for addresses

Definition at line 84 of file types.h.

B.19.3.10. typedef int32_t error_t

Data type for errors

Definition at line 86 of file types.h.

B.19.3.11. typedef uint32_t version_t

Data type for **driver** (p. 24) versions

Definition at line 88 of file types.h.

B.19.3.12. typedef signed int ssize_t

Data type for sizes

Definition at line 90 of file types.h.

B.19.3.13. typedef int32_t thread_handler

The thread handler is just an ID; this value is used as an offset into the TCB array.

Definition at line 96 of file types.h.

B.19.3.14. typedef struct thread* threadptr

Thread pointer

Definition at line 98 of file types.h.

B.19.3.15. typedef struct thread_control_block_t tcb_t

Thread control block

Definition at line 100 of file types.h.

B.19.3.16. typedef uint32_t sched_t

Scheduling information

Definition at line 106 of file types.h.

B.19.3.17. typedef int32_t cc_spinlock_t

Spinlock variable (busy waiting) for synchronisation

Definition at line 118 of file types.h.

B.19.3.18. typedef struct pthread_mutex pthread_mutex_t

B.19.3.19. typedef struct pthread_cond pthread_cond_t

B.19.3.20. typedef struct mem_cfg_data memory_t

B.19.3.21. typedef thread_handler pthread_t

POSIX thread

Definition at line 176 of file types.h.

B.19.4. Enumeration Type Documentation

B.19.4.1. enum bool_t

Boolean data type

Enumerator:

FALSE

TRUE

Definition at line 76 of file types.h.

B.19.4.2. enum sched_policy

Scheduling policy

Enumerator:

SCHED_NONE

SCHED_HRT

SCHED_NRT

Definition at line 108 of file types.h.

B.20. uart.h File Reference

Functions for sending/receiving from/to UART (Universal Asynchronous Receiver Transmitter).

```
#include "clock.h"
```

Defines

- #define **UART_BASE_ADDRESS** 0xF0001000
- #define **UART_rxdata_offset** 0x0
- #define **UART_txdata_offset** 0x4
- #define **UART_status_offset** 0x8
- #define **UART_control_offset** 0xC
- #define **UART_divisor_offset** 0x10
- #define **UART_RX_DATA** *((volatile char *) (UART_BASE_ADDRESS + UART_rxdata_offset))
- #define **UART_TX_DATA** *((char *) (UART_BASE_ADDRESS + UART_txdata_offset))
- #define **UART_STATUS** *((volatile char *) (UART_BASE_ADDRESS + UART_status_offset))
- #define **UART_CONTROL** *((volatile char *) (UART_BASE_ADDRESS + UART_control_offset))
- #define **UART_DIVISOR** *((unsigned short *) (UART_BASE_ADDRESS + UART_divisor_offset))
- #define **FE_bit_idx** 1
- #define **BRK_bit_idx** 2
- #define **ROE_bit_idx** 3
- #define **TOE_bit_idx** 4
- #define **TMT_bit_idx** 5
- #define **TRDY_bit_idx** 6
- #define **RRDY_bit_idx** 7
- #define **E_bit_idx** 8
- #define **DCTS_bit_idx** 9
- #define **CTS_bit_idx** 10
- #define **RTS_bit_idx** 11
- #define **UART_NEW_CHARACTER_RECEIVED** (UART_STATUS & (1 << RRDY_bit_idx))
- #define **UART_TRANSMITTER_READY** (UART_STATUS & (1 << TRDY_bit_idx))

Functions

- void **set_baud_rate** (unsigned int baud_rate)
- void **set_uart_divisor** (unsigned short divisor)
- int **is_new_character_fully_received** (void)
- int **is_transmitter_ready** (void)
- void **uart_put_char** (char c)
- char **uart_get_char** (void)
- void **uart_put_string** (char *s)
- void **uart_put_chars** (char *data, int n)
- void **uart_get_chars** (char *data, int n)
- void **uart_get_string** (char *buffer, unsigned char maxLen)
- void **uart_send_byte_as_hex_string** (char byte)
- void **uart_send_short_as_hex_string** (short sh)
- void **uart_send_int_as_hex_string** (int i)
- void **uart_send_long_as_hex_string** (long long l)

B.20.1. Detailed Description

Definition in file **uart.h**.

B.20.2. Define Documentation

B.20.2.1. #define UART_BASE_ADDRESS 0xF0001000

Definition at line 12 of file uart.h.

B.20.2.2. #define UART_rxdata_offset 0x0

Definition at line 17 of file uart.h.

B.20.2.3. #define UART_txdata_offset 0x4

Definition at line 18 of file uart.h.

B.20.2.4. #define UART_status_offset 0x8

Definition at line 19 of file uart.h.

B.20.2.5. #define UART_control_offset 0xC

Definition at line 20 of file uart.h.

B.20.2.6. #define UART_divisor_offset 0x10

Definition at line 21 of file uart.h.

B.20.2.7. #define UART_RX_DATA *((volatile char *) (UART_BASE_ADDRESS + UART_rxdata_offset))

Definition at line 24 of file uart.h.

B.20.2.8. #define UART_TX_DATA *((char *) (UART_BASE_ADDRESS + UART_txdata_offset))

Definition at line 25 of file uart.h.

B.20.2.9. #define UART_STATUS *((volatile char *) (UART_BASE_ADDRESS + UART_status_offset))

Definition at line 26 of file uart.h.

B.20.2.10. #define UART_CONTROL *((volatile char *) (UART_BASE_ADDRESS + UART_control_offset))

Definition at line 27 of file uart.h.

B.20.2.11. #define UART_DIVISOR *((unsigned short *) (UART_BASE_ADDRESS + UART_divisor_offset))

Definition at line 28 of file uart.h.

B.20.2.12. #define FE_bit_idx 1

Definition at line 32 of file uart.h.

B.20.2.13. #define BRK_bit_idx 2

Definition at line 33 of file uart.h.

B.20.2.14. #define ROE_bit_idx 3

Definition at line 34 of file uart.h.

B.20.2.15. #define TOE_bit_idx 4

Definition at line 35 of file uart.h.

B.20.2.16. #define TMT_bit_idx 5

Definition at line 36 of file uart.h.

B.20.2.17. #define TRDY_bit_idx 6

Definition at line 37 of file uart.h.

B.20.2.18. #define RRDY_bit_idx 7

Definition at line 38 of file uart.h.

B.20.2.19. #define E_bit_idx 8

Definition at line 39 of file uart.h.

B.20.2.20. #define DCTS_bit_idx 9

Definition at line 40 of file uart.h.

B.20.2.21. #define CTS_bit_idx 10

Definition at line 41 of file uart.h.

B.20.2.22. #define RTS_bit_idx 11

Definition at line 45 of file uart.h.

B.20.2.23. #define UART_NEW_CHARACTER_RECEIVED (UART_STATUS & (1 << RRDY_bit_idx))

Tests whether a new character is fully received via UART. If it is received completely, the return value is not equal to 0. – When a new character is fully received via the RxD input, it is transferred into the rxdata register, and the status register's rrdy bit is set to 1. – REMARK: When software reads a value from the rxdata register, the status register's rrdy bit is set to 0 by hardware.

Definition at line 58 of file uart.h.

B.20.2.24. #define UART_TRANSMITTER_READY (UART_STATUS & (1 << TRDY_bit_idx))

Tests whether the transmitter is ready (return value 0 means "not ready") for sending a new character. Characters should not be written to the txdata register until the transmitter is ready for a new character, as indicated by the trdy bit in the status register.

Definition at line 67 of file uart.h.

B.20.3. Function Documentation

B.20.3.1. void set_baud_rate (unsigned int *baud_rate*)

This function can be used to modify the baud rate to a specific value (the UART device works with any of the following standard baud rates for RS-232 connections: 600, 1200, 2400, 4800, 9600, 14400, 19200, 38400, 57600, 115200. Accordingly, you should only use these values as a parameter of the function). The default baud rate is set to 115200 baud, also after each start of the FPGA board. Also make sure that the constant CLOCK_FREQUENCY_MHZ in file clock.h is set to the correct value of the current CPU clock frequency.

B.20.3.2. void set_uart_divisor (unsigned short *divisor*) [inline]

Writes divisor in 16-bit divisor register of the UART core hardware(sets the baud rate). – Baud rate and divisor values are related: baud_rate = (CLOCK_FREQUENCY_MHZ/(divisor + 1)) divisor = (short)((CLOCK_-FREQUENCY_MHZ * 1000000) / desired_baud_rate + 0.5) – Remark: The baud rate is calculated based on the CPU clock frequency. Accordingly you need to adjust the constant: CLOCK_FREQUENCY_MHZ in the file: clock.h

B.20.3.3. int is_new_character_fully_received (void) [inline]

Tests whether a new character is fully received via UART. If it is received completely, the return value is not equal to 0. – When a new character is fully received via the RxD input, it is transferred into the rxdata register, and the status register's rrdy bit is set to 1. – REMARK: When software reads a value from the rxdata register, the status register's rrdy bit is set to 0 by hardware.

B.20.3.4. int is_transmitter_ready (void) [inline]

Tests whether the transmitter is ready (return value 0 means "not ready") for sending a new character. Characters should not be written to the txdata register until the transmitter is ready for a new character, as indicated by the trdy bit in the status register.

B.20.3.5. void uart_put_char (char *c*) [inline]

Sends a char (one byte) over UART; blocking until the transmitter is ready.

B.20.3.6. char uart_get_char (void) [inline]

Receives a char (one byte) over UART; blocking until a new character is fully received.

B.20.3.7. void uart_put_string (char * *s*)

Sends a string over UART. The number of chars is detected by the " char at the end of the string. Blocking until the whole string is sent. Uses the function uart_put_char.

B.20.3.8. void uart_put_chars (char * *data*, int *n*)

Sends n chars from the array data. blocking, until all n chars are sent. Uses the function uart_put_char.

B.20.3.9. void uart_get_chars (char * *data*, int *n*)

Receives n chars over UART und writes them into a buffer at data. This buffer should have been created in advance. Blocking, until all n chars are fully received. Uses the function uart_get_char.

B.20.3.10. void uart_get_string (char * *buffer*, unsigned char *maxLen*)

Receives chars in the char array buffer until either the array is full (maxLen-1) or the line end char ('\n') is received. Blocking, until all chars are fully received. Uses the function uart_get_char.

B.20.3.11. void uart_send_byte_as_hex_string (char *byte*)

Converts a byte to hexadecimal string and sends it via UART.

B.20.3.12. void uart_send_short_as_hex_string (short *sh*)

Converts a short to hexadecimal string and sends it via UART.

B.20.3.13. void uart_send_int_as_hex_string (int *i*)

Converts a int value to hexadecimal string and sends it via UART.

B.20.3.14. void uart_send_long_as_hex_string (long long *l*)

Converts a long value to hexadecimal string and sends it via UART.

B.21. unistd.h File Reference

Generic read and write operations.

`#include <sys/types.h>`

Functions

- int **close** (int d)
- **ssize_t read** (int d, void *buf, size_t nbytes)
- **ssize_t write** (int d, const void *buf, size_t nbytes)

B.21.1. Detailed Description

Definition in file **unistd.h**.

B.21.2. Function Documentation

B.21.2.1. int close (int *d*)

Delete a descriptor.

The **close()** (p. 121) system call deletes a descriptor from the per-process object reference table. If this is the last reference to the underlying object, the object will be deactivated.

Returns:

The **close()** (p. 121) function returns the value 0 if successful; otherwise the value -1 is returned and the global variable errno is set to indicate the error.

B.21.2.2. ssize_t read (int *d*, void * *buf*, size_t *nbytes*)

Read input.

The **read()** (p. 121) system call attempts to read nbytes of data from the object referenced by the descriptor d into the buffer pointed to by buf.

Returns:

If successful, the number of bytes actually read is returned. Upon reading end-of-file, zero is returned. Otherwise, a -1 is returned and the global variable errno is set to indicate the error.

B.21.2.3. ssize_t write (int *d*, const void * *buf*, size_t *nbytes*)

Write output.

The **write()** (p. 121) system call attempts to write nbytes of data to the object referenced by the descriptor d from the buffer pointed to by buf.

Returns:

Upon successful completion the number of bytes which were written is returned. Otherwise a -1 is returned and the global variable errno is set to indicate the error.

B.22. vo.h File Reference

Definitions of memory regions for virtual output.

Defines

- #define **PER_VIO** (0xe0010000)
- #define **VO_ADDRESS** (PER_VIO + 0x04)
- #define **VO_WORD** (PER_VIO + 0x08)
- #define **VO_DBL_D4C** (PER_VIO + 0x38)
- #define **VO_DBL_X4C** (PER_VIO + 0x40)
- #define **VIO_PREFIX** "%#"
- #define **P64**(a) (∗((**uint64_t** volatile ∗) a))

B.22.1. Detailed Description

Definition in file **vo.h**.

B.22.2. Define Documentation

B.22.2.1. #define PER_VIO (0xe0010000)

The base address for the virtual output.

Definition at line 44 of file vo.h.

B.22.2.2. #define VO_ADDRESS (PER_VIO + 0x04)

Bytes written to this address are put into the virtual output.

Definition at line 47 of file vo.h.

B.22.2.3. #define VO_WORD (PER_VIO + 0x08)

Write one word to STDOUT.

Definition at line 50 of file vo.h.

B.22.2.4. #define VO_DBL_D4C (PER_VIO + 0x38)

Special log, hi is interpreted as chars, lo as word dec.

Definition at line 53 of file vo.h.

B.22.2.5. #define VO_DBL_X4C (PER_VIO + 0x40)

Special log, hi is interpreted as chars, lo as word hex.

Definition at line 56 of file vo.h.

B.22.2.6. #define VIO_PREFIX "%#"

The prefix allows easy filtering of VIO output in console mode.

Definition at line 59 of file vo.h.

B.22.2.7. #define P64(a) (*((uint64_t volatile *) a))

Fast access to a 64-bit value.

Definition at line 62 of file vo.h.

References

[1] Cassé, H., and Sainrat, P. OTAWA, a framework for experimenting WCET computations. In *3rd European Congress on Embedded Real-Time Software* (2006).

[2] FAQ of comp.realtime. http://www.faqs.org/faqs/realtime-computing/faq/, July 1998. visited November 2009.

[3] Infineon Technologies AG. *TriCore 1 Architecture Volume 1: Core Architecture V1.3 & V1.3.1*, Jan. 2008.

[4] Lea, D. A memory allocator. http://g.oswego.edu/, 2000.

[5] Masmano, M., Ripoll, I., Crespo, A., and Real, J. TLSF: A New Dynamic Memory Allocator for Real-Time Systems. In *ECRTS '04: Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS'04)* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 79–86.

[6] Mische, J., Guliashvili, I., Uhrig, S., and Ungerer, T. How to Enhance a Superscalar Processor to Provide Hard Real-Time Capable In-Order SMT. In *23rd International Conference on Architecture of Computing Systems (ARCS 2010), Proceedings* (Hannover, Germany, Feb. 2010), Springer-Verlag, pp. 2–14.

[7] IEEE Std 1003.1, 2004 Edition. The Open Group Base Specifications Issue 6, 2004.

[8] QNX Neutrino RTOS. http://www.qnx.com/products/. Visited November 2009.

[9] Ungerer, T., Cazorla, F., Sainrat, P., Bernat, G., Petrov, Z., Rochange, C., Quinones, E., Gerdes, M., Paolieri, M., and Wolf, J. MERASA: Multi-core execution of hard real-time applications supporting analysability. *IEEE Micro 99*, PrePrints (2010).

[10] Wolf, J., Gerdes, M., Kluge, F., Uhrig, S., Mische, J., Metzlaff, S., Rochange, C., Cassé, H., Sainrat, P., and Ungerer, T. RTOS Support for Parallel Execution of Hard Real-Time Applications on the MERASA Multi-Core Processor. In *Proceedings of the 13th IEEE International Symposium on Object/component/service-oriented Real-time Distributed Computing (ISORC 2010)* (Carmona, Spain, May 2010), IEEE Computer Society, pp. 193–201.